

Designing Systems on FPGAs

Document Version 1.2, April 2008

Software, documentation and related materials:

Copyright © 2008 Altium Limited.

All rights reserved. You are permitted to print this document provided that (1) the use of such is for personal use only and will not be copied or posted on any network computer or broadcast in any media, and (2) no modification of the document is made. Unauthorized duplication, in whole or part, of this document by any means, mechanical or electronic, including translation into another language, except for brief excerpts in published reviews, is prohibited without the express written permission of Altium Limited. Unauthorized duplication of this work may also be prohibited by local statute. Violators may be subject to both criminal and civil penalties, including fines and/or imprisonment.

Altium, Altium Designer, Board Insight, CAMtastic, CircuitStudio, Design Explorer, DXP, LiveDesign, NanoBoard, NanoTalk, Nexar, nVisage, P-CAD, Protel, SimCode, Situs, TASKING, and Topological Autorouting and their respective logos are trademarks or registered trademarks of Altium Limited or its subsidiaries.

Microsoft, Microsoft Windows and Microsoft Access are registered trademarks of Microsoft Corporation. OrCAD, OrCAD Capture, OrCAD Layout and SPECCTRA are registered trademarks of Cadence Design Systems Inc. AutoCAD is a registered trademark of Autodesk Inc. HP-GL is a registered trademark of Hewlett Packard Corporation. PostScript is a registered trademark of Adobe Systems, Inc. All other registered or unregistered trademarks referenced herein are the property of their respective owners and no trademark rights to the same are claimed.

1	Designing a System to Display Video	1-5
1.1	The nuts and bolts of capturing video	1-5
1.2	The nuts and bolts of Displaying video on a touch screen.....	1-6
1.3	A high level view of our system	1-6
2	Utilizing microprocessor softcores	2-7
2.1	How soft is a softcore?	2-7
2.2	Using a softcore in an FPGA schematic	2-8
2.3	Accessing softcore datasheets	2-8
2.4	The TSK3000 Processor	2-8
2.5	Exercise 1 – Placing a softcore onto an FPGA schematic.....	2-9
2.6	Configuring the TSK3000 processor.....	2-9
2.7	Exercise 2 – Configuring the TSK3000A Processor	2-11
3	Wishbone Interconnection Architecture	3-12
3.1	Wishbone compatible cores	3-12
3.2	Wishbone interface signal descriptions.....	3-13
3.3	Pin naming conventions for wishbone signal pins	3-15
4	Placing the FPGA Peripherals	4-17
4.1	Exercise 3 – Placing the peripherals	4-17
5	Bus Interconnectivity and Arbitration	5-20
5.1	Wishbone Interconnect Component WB_INTERCON.....	5-20
5.2	Configuring the Wishbone Interconnect component.....	5-21
5.3	Exercise 4 – Connecting our Peripherals.....	5-24
5.4	Managing memory access	5-25
5.5	Arbitration of multiple masters sharing a single slave.....	5-25
5.6	Wishbone Dual- and Multi- Master devices.....	5-26
5.7	Configuring the wishbone multi- master component.....	5-27
5.8	Exercise 5 – Defining our Multi Master Components.....	5-30
6	Interfacing to External Memory	6-31
6.1	Configuring the SRAM Controller	6-31
6.2	The Shared Memory Controller	6-33
6.3	Configuring the Shared Memory Controller.....	6-34
6.4	Exercise 6 – Placing and configuring memory controllers	6-36
6.5	Exercise 7 – Multiple memories as slaves	6-37
6.6	Exercise 8 – Placing the Port Plug In Components.....	6-38
6.7	Exercise 10 – Wiring Up Your FPGA Design	6-40
7	Configuring Processor Memory	7-42
7.1	Dividing the processor memory	7-43
7.2	Exercise 11 – Configuring the processor memory.....	7-45
7.3	Configuring processor peripherals.....	7-46
7.4	Exercise 12 – Specifying processor peripherals.....	7-47
8	Finalizing the Design, Compiling and Resolving Errors.....	8-48
9	Creating Embedded Systems on FPGAs	9-49
9.1	Exercise 13 – Editor Basics.....	9-49
9.2	The TASKING tool chain	9-53
9.3	The Build flow	9-53
9.4	Targeting the Project.....	9-54
9.5	Project options.....	9-54
9.6	Attaching an embedded project to an FPGA project	9-66
9.7	Exercise 13 – Linking projects.....	9-68
9.8	Exercise 14 – Building our design	9-68
9.9	Exercise 15 – Updating our software Live.....	9-70
10	Updating the Design to Display Video	10-71
10.1	Defining Sections.....	10-71
10.2	Exercise 16 – Updating our software Live.....	10-72

10.3	Exercise 17 – Updating our Software to Capture Video	10-73
10.4	Utilizing the Virtual Instruments	10-74
10.5	Using instruments to control our design.....	10-74
10.6	Exercise 18 – Updating the Wishbone Port IO	10-75
10.7	Exercise 19 – Adding the Virtual Instruments	10-76
10.8	Exercise 20 - Rebuilding the project with instruments	10-78
10.9	Updating our embedded project to use the instruments	10-78
10.10	Exercise 21 – Adding Additional Source Files.....	10-78
10.11	Exercise 22 – Updating main.c.....	10-78
10.12	Exercise 23 – Controlling our design with instruments	10-79
11	Real-time Debugging of a Processor	11-81
11.1	On-Chip Debugging	11-81
11.2	A word about simulation.....	11-81
11.3	Embedded control panels.....	11-83
11.4	Instrument Rack – Nexus Debugger.....	11-89
12	C to Hardware.....	12-92
12.1	What are the benefits of the C-to-Hardware Compiler?.....	12-92
12.2	Using the CHC Compiler	12-92
12.3	Implementing C to Hardware in our Design	12-93
12.4	Exercise 24 – Making Room for C to Hardware	12-93
12.5	Exercise 25 – Adding the Application Specific Processor.....	12-94
12.6	Exercise 26 – Giving the ASP Access to Memory (part 1).....	12-95
12.7	Exercise 27 – Giving the ASP Access to Memory (part 2).....	12-97
12.8	Configuring the ASP	12-101
12.9	Exercise 28 – Accelerating our performance with ASP	12-103
13	OpenBus.....	13-106
13.1	Creating an OpenBus version of our design	13-106
13.2	Exercise 29 – Beginning an FPGA Project Using OpenBus	13-106
13.3	OpenBus Devices	13-107
13.4	Exercise 30 – Placing Open Bus Components (part 1)	13-107
13.5	Exercise 31 – Using an Interconnect	13-111
13.6	Exercise 32 – Linking OpenBus Ports	13-114
13.7	Exercise 33 – Configuring our processor under open bus.....	13-115
13.8	Exercise 34 – Configuring the GPIO Component.....	13-115
13.9	Exercise 35 – Finalizing the Interconnect Component	13-116
13.10	Defining the Memory Side of our System.....	13-117
13.11	Exercise 36 – Adding the Remaining Components	13-117
13.12	Exercise 37 – Completing the Connectivity.....	13-118
13.13	Exercise 38 – Configuring the SRAM Controllers.....	13-118
13.14	Exercise 39 – Configuring the Arbiters	13-120
13.15	Exercise 40 – Configuring the Interconnects.....	13-121
13.16	Finalizing the OpenBus Portion of the Design.....	13-121
13.17	Exercise 41 – Configuring the Processor Memory	13-122
13.18	Exercise 42 - Creating a sheet symbol from an OpenBus Document	13-123
13.19	Exercise 43 – Linking an Embedded Project to an OpenBus Design.....	13-124
13.20	Exercise 44 – Processing an OpenBus Design.....	13-125
14	Review	14-126

1 Designing a System to Display Video

This next section of the course is designed to take you through the process of developing a complete soft processor based system on an FPGA. To accomplish this, we will follow a single design through a variety of different stages and different capture methods. The design we've chosen is a simple video display system capable of capturing composite video and displaying the video on a touch screen, with options to scale and rotate the image.

Central to our design will be the TSK3000A, a derivative of the TSK3000 family of soft microprocessors supplied with Altium Designer. Supporting the TSK3000A will be a host of peripherals that will supply the means with which to both capture and display our video on the touch screen located on the Desktop NanoBoard.



1.1 The nuts and bolts of capturing video

The Desktop NanoBoard's peripheral board PB01 supports the connection of external PAL or NTSC analog video signals. Both Composite Video and S-Video inputs are supported.

- Connection of a Composite Video signal is made through two RCA phono jacks, designated J5 and J6, which cater for the Luma (luminance or intensity) and Chroma (chrominance or color) components of the signal respectively.

The analog video input is converted into an 8-bit digital YCbCr 4:2:2 component video through the use of a TVP5150AM1 video decoder device from Texas Instruments.

The decoder provides a 9-bit ADC, with sampling carried out in accordance with the ITU-R BT.601 recommendation. This recommendation defines the scheme for encoding interlaced analog video signals – for both 625 and 525 line systems – in *digital* form.

The converted digital video stream can be output in one of two formats:

- 8-bit ITU-R BT.656 interface with embedded synchronization
- 8-bit 4:2:2 YCbCr, with discrete synchronization signals

By default the TVP5150AM1 is configured to output digital video in the ITU-R BT.656 format and this is the format expected by a BT656 Controller used within an FPGA design.

In our design, **we will use the BT656 Controller core** supplied with Altium Designer to fetch the video from the video decoder and supply that to the rest of our system.

The BT656 Controller takes as its inputs from the TVP5150AM1:

- VIDIN_DATA[7..0] – the ITU-R BT.656-compliant video stream
- VIDIN_PCLK – the pixel clock, fixed at 27 MHz and used to clock the output video data stream
- VIDIN_INTERQ_GPLC – used to indicate the vertical blanking interval of the video data stream

The video decoder is controlled by a processor within the FPGA design over the I2C bus. The decoder (an I2C slave) is **controlled through an intermediate I2C controller** (the I2C master, also resident within the FPGA design).

1.2 The nuts and bolts of Displaying video on a touch screen

The Desktop NanoBoard provides high quality color display through a Hitachi TX09D50VM1CAA TFT (Thin Film Transistor) LCD panel. The panel also features an analog resistive touch screen. From a user perspective you can either output straightforward graphics to the panel, or provide a more sophisticated level of interaction involving input to your FPGA design from the touch screen.

The 3.5" panel features a Transmissive Color TFT LCD, with a resolution of 240(W) by 340(H). The display is based on an active matrix and backlight for the panels is provided through 6 LEDs. The panel is fixed to use 16bpp (bits per pixel) display, with bit 0 from both Red and Blue intensity inputs tied to GND. This results in a total of 65536 colors being available for display.

In our design, **we will use the WB_VGA Configurable Wishbone Display Driver** supplied with Altium Designer to fetch 16bpp-formatted data from external memory, process it, and display it on the connected TFT LCD panel. All of the control signals are generated in-core, using an integrated TFT Timing Controller unit.

1.3 A high level view of our system

It helps to have a high-level understanding of how our system will go together before getting started. Below is a block diagram that explains the flow of data in to and out of our system.

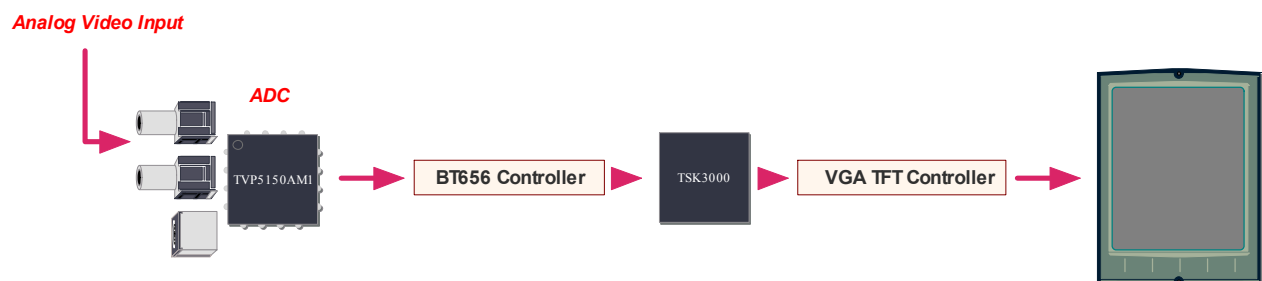


Figure 1. Simplified flow diagram of our system.

Figure 1 shows a simplified flow diagram of our system. On the left hand side of the diagram you can see the analog video input. This is converted to digital by the TVP5150AM1 and output as BT656 to the FPGA-based BT656 decoder. This will then be processed by the TSK3000 and output to the touch screen display using an FPGA-based VGA controller.

2 Utilizing microprocessor softcores

In this section we will begin to explore utilizing soft microprocessor cores on FPGAs.

2.1 How soft is a softcore?

Altium Designer comes bundled with a number of pre-verified, pre-synthesized microprocessor softcores that can be incorporated into an FPGA project. These processors are labeled as *soft* because they are implemented as a downloadable core (utilizing resources on the FPGA) that runs from an FPGA device rather than as a *hard* physical microprocessor in its own distinct package.

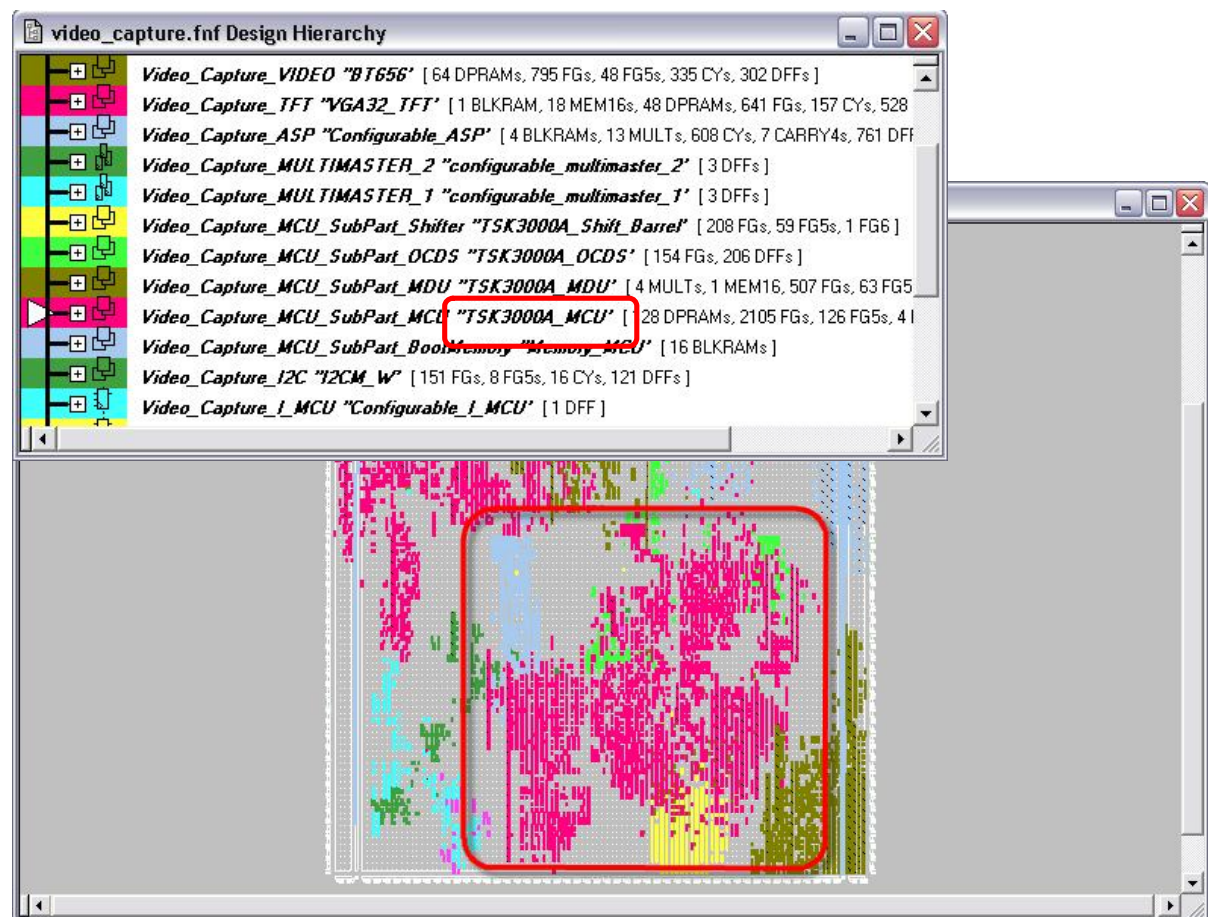


Figure 2. Pictorial representation of a softcore loaded onto an FPGA device.

One of the fundamental differences between FPGA-based microprocessor cores (such as those bundled with Altium Designer) and hard processors is the relationship of the peripherals to the processor itself. In traditional hard processors, a set of standard peripherals will come “on-chip” and are fixed (i.e. cannot be changed). An example might be an 8051 with an on-chip High Speed USB interface. FPGA-based processors on the other hand, generally include only the CPU enabling the engineer to pack whatever peripherals might be required ‘around’ the processor. This is central to the increased flexibility / scalability of FPGA-based “systems”. Systems can be modified in the soft domain to add and remove peripherals as a design changes, allowing more flexibility at the design stage to make real-time modifications to a system thus not requiring the engineer to “lock-in” the hardware up front.

2.2 Using a softcore in an FPGA schematic

Building an FPGA project that incorporates a softcore is no different from building any other FPGA project; you simply select components from a library and place them onto your schematic. The FPGA softcores can be accessed from the integrated library `FPGA Processors.IntLib`

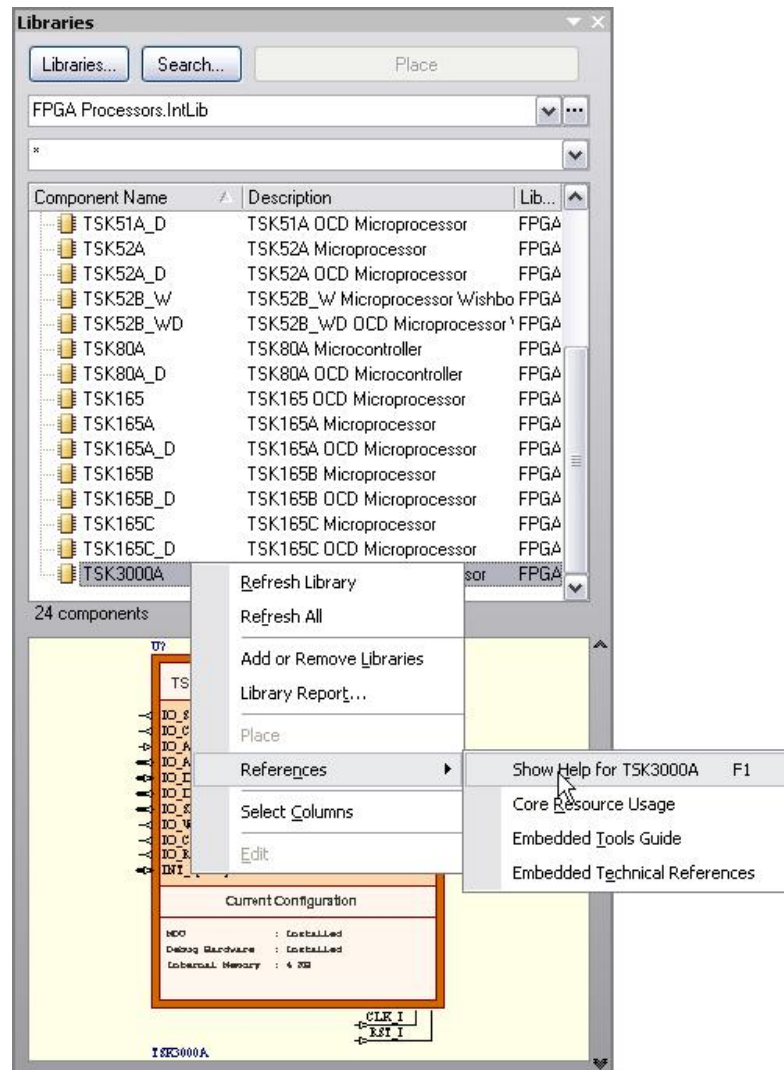


Figure 3. FPGA Processors.IntLib library contents with popup Help

2.3 Accessing softcore datasheets

Detailed data sheets can be accessed for each device by highlighting the device and selecting the **F1** key or by right-clicking the device in the **Libraries** panel and selecting **Show Help for <Device Name>** as seen in Figure 3.

2.4 The TSK3000 Processor

Central to our design is the TSK3000A core. The TSK3000A is a configurable 32-bit, Wishbone-compatible, RISC processor core. Most instructions are 32-bits wide and execute in a single clock cycle. In addition to fast register access, the TSK3000A features a user definable (from 1K Bytes up to 1M Bytes) amount of zero-wait state block RAM, with true dual-port access.

The TSK3000A also features:

- FPGA device-independent implementation
- 5-stage pipelined RISC processor
- 32x32- to 64-bit hardware multiplier, signed and unsigned
- 32x32-bit hardware divider
- 32-bit single-cycle barrel shifter
- 32 input interrupts, individually configurable to be level or edge sensitive and used in one of two modes:
 - Standard Mode - all interrupts jump to the same, configurable base vector
 - Vectored Mode - providing 32 vectored priority interrupts, each jumping to a separate interrupt vector
- Internal Harvard architecture with simplified external memory access
- 4GByte address space
- Wishbone I/O and memory ports for simplified peripheral connection
- Full Viper-based software development tool chain – C compiler/assembler/source-level debugger/profiler
- C-code compatible with other Altium Designer 8-bit and 32-bit Wishbone-compliant processor cores, for easy design migration

2.5 Exercise 1 – Placing a softcore onto an FPGA schematic

In this exercise we will create a new FPGA project and add the TSK3000A to a new, blank schematic document.

1. Create a new FPGA project and a new schematic sheet
2. Adjust the sheet to C-size within the **Document Options** dialog located under **Design»Document Options** or by double clicking in the border area around the outside of the schematic document.
3. Save the FPGA project using **File»Save Project As...** and you will be prompted to save both the schematic and the FPGA project. Name the files `CHC_Image_Rotation.SchDoc` and `CHC_Image_Rotation.PrjFpg` respectively.
4. Select the TSK3000A processor from the FPGA Processors.IntLib library and place it onto the schematic.
5. Annotate the component U2 and save your work.

2.6 Configuring the TSK3000 processor

The **Configure (32-bit Processors)** dialog (Figure 4) is accessible by right-clicking the processor in the schematic and selecting **Configure...** or alternatively by selecting the **Configure** button in the component properties dialog.

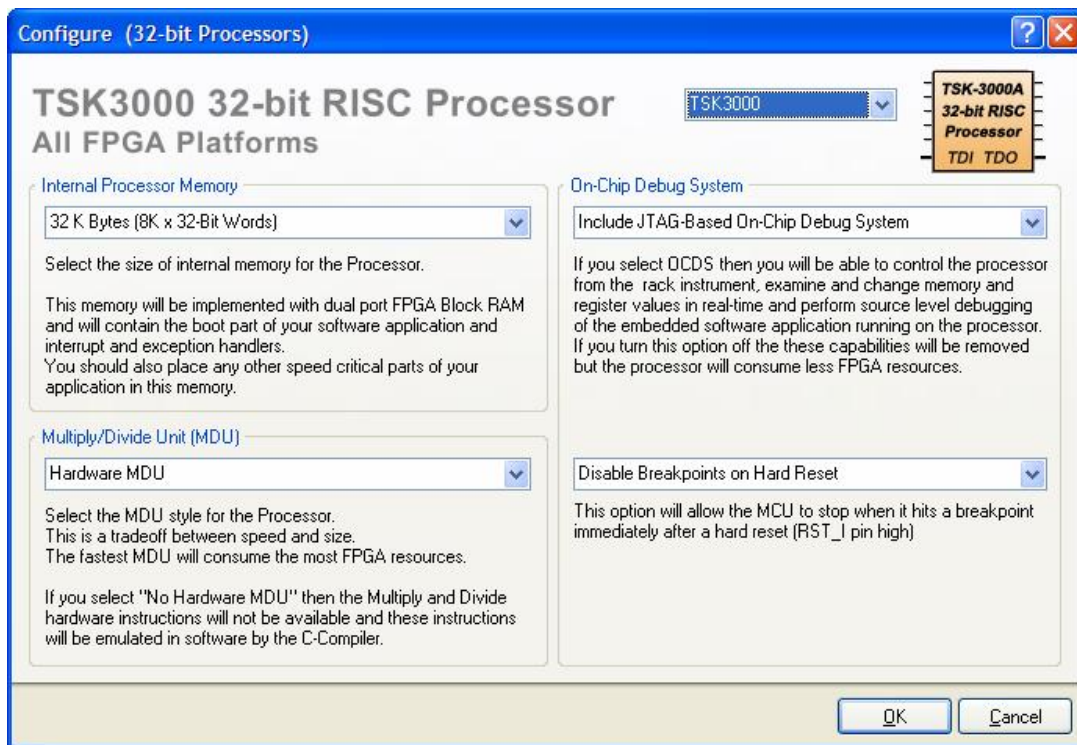


Figure 4. Processor configuration dialog

Several options are available when configuring the processor:

- **Internal Processor Memory**

This is a user configurable feature of the TSK3000A. Utilizing dual-port FPGA Block RAM, this area generally contains the boot portion of your application as well as interrupt and exception handlers.

This memory is generally the fastest and you will want to consider placing any speed-critical parts of your application in this memory space. This memory is however limited to the amount of Dual Port Block RAM available in the FPGA and thus the amount you specify needs to be balanced for the device selected.

- **The Multiply / Divide Unit (MDU)**

Fundamentally there are two approaches to managing multiply / divide instructions, one being to implement them in hardware using FPGA resources to create the multiply (MULT, MULTU) and divide (DIV, DIVU) hardware instructions, or alternatively implementing them in software such that these instructions are emulated by the C-Compiler.

This is a tradeoff of speed versus size. The Hardware MDU will be faster as these instructions have dedicated hardware resources, however it will consume FPGA resources and thus the space available in the FPGA will be reduced.

- **On-Chip Debug System**

The TSK3000 includes an optional on-chip debugging (OCD) system. The OCD system can be used to debug a processor as it executes in system, in real time. This includes options to:

- Reset, Go, Halt processor control
- Single or multi-step debugging
- Read-write access for internal processor registers
- Read-write access for memory and I/O space
- Unlimited software breakpoints

The on chip-debugging system can be removed at any time which will then free up the FPGA resources it requires. No changes will be required in the source embedded

application when switching between OCD and non-OCD versions of the microprocessor.

- **Breakpoints on Reset**

The TSK3000A has the feature to enable the application to run to a breakpoint after a hard reset (RST_I pin goes high). This option is user configurable and can be selected depending on the needs of the target application.

2.7 Exercise 2 – Configuring the TSK3000A Processor

Having placed the TSK3000A in our design, we now need to configure it for our application.

1. Right-click the TSK3000A in the schematic and select **Configure U2 (TSK3000A)**. This will launch the Configure (32-bit Processors) dialog.
2. Set the **Internal Processor Memory** option to **32K Bytes (8K x 32-Bit Words)**
3. Set the **Multiply/Divide Unit (MDU)** option to **Hardware MDU**
4. Set the **On-Chip Debug System** option to **Include JTAG-Based On-Chip Debug System**
6. Set the **Breakpoints on Reset** option to **Disable Breakpoints on Hard Reset**

3 Wishbone Interconnection Architecture

Altium Designer makes extensive use of the Wishbone Interconnection Architecture. A copy of the Wishbone standard is available from www.opencores.org or located within the help system in Altium Designer. Simply stated, Wishbone is an open-source standard that specifies the *interface* to IP cores, thus making it easier to connect blocks of Intellectual Property cores together in an FPGA / ASIC design. The intent being that if all IP blocks have a standard interface, blocks of IP will then be more easily reusable, more easily interchangeable, and the time spent negotiating transactions between cores will be reduced or eliminated.

What Wishbone Is Not

Wishbone does not specify the behavior or architecture of the IP itself. Instead, Wishbone is intended to be “General Purpose” with its emphasis on data exchange and not a core’s function.

Like discrete microprocessors which include an address bus, data bus, and handshaking lines, the Wishbone Interconnection Architecture defines the signals required to interface IP blocks to one another and how they operate. Wishbone does not imply the electrical characteristics of the interconnections but is intentionally focused on the logical implementation of a common mechanism for the interaction between IP.

3.1 Wishbone compatible cores

With the TSK3000A configured, the next stage in our project is to specify the peripheral cores required and connect them to the processor. There is a range of Wishbone compatible peripheral cores provided in the `FPGA_Peripherals.IntLib` library and it’s this Wishbone compatibility that makes the interconnection of these components fast and easy.

For example, in the figure below, notice the uniformity of the various interfaces of these two components.

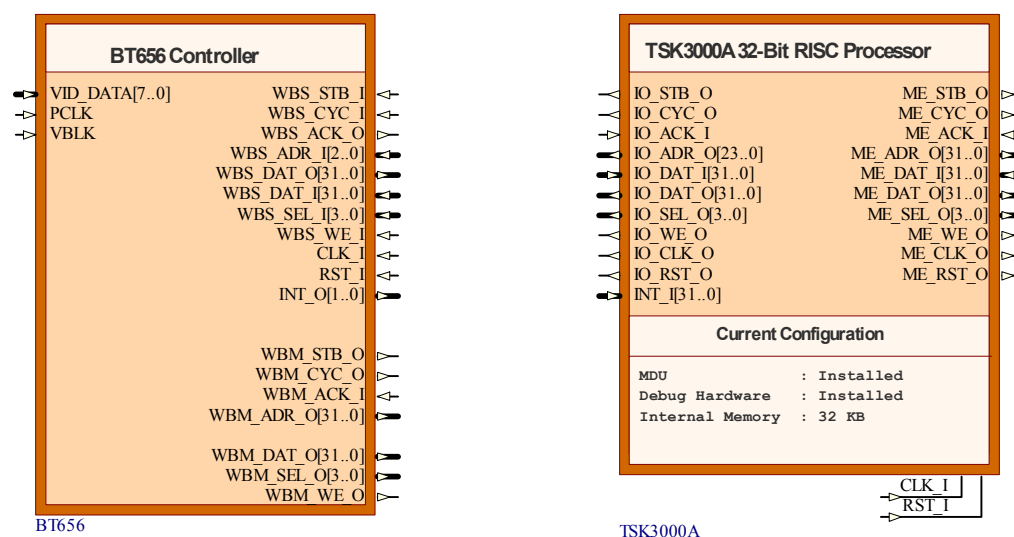


Figure 5. A look at the Wishbone signal pins on schematic symbols

The pin names are quite similar as they are derived from the Wishbone standard and their names reference the Wishbone interface signal names. To further simplify the interfacing of IP 'components' to one another, Wishbone requires that all Wishbone signals are active high. This makes representing the signals in schematic capture tools easier as it eliminates the challenge of uniformly representing active low logic (graphically) in a schematic design (i.e. over bar, hash marks, slashes, etc.)

3.2 Wishbone interface signal descriptions

The wishbone signals can be grouped as either Master, Slave, common to both Slave and Master, or SYSCON module signals (a module is SYSCON if it drives the system clock CLK_O and the reset RST_O signals). The following table is a list of Wishbone signals used in Altium Designer. For a complete list of Wishbone signals and signal descriptions, refer to the Wishbone specification found at www.opencores.org or in the help system.

Wishbone SYSCON Signals

Signal Name	Signal Description
CLK_O	External (system) clock signal (identical to CLK_I) made available for connecting to the CLK_I input of a slave device
RST_O	Reset signal available to connect to a slave device

Wishbone Common Signals (Master and Slave)

Signal Name	Signal Description
CLK_I	External (system) clock signal
DAT_I[x..y]	Data input array from an external device such as memory. Maximum width 64 bits.
DAT_O[x..y]	Data output array to an external device such as memory. Maximum width 64 bits.
RST_I[x..y]	External (system) reset signal

Wishbone Master Signals

Signal Name	Signal Description
ACK_I	Standard Wishbone acknowledgement signal used to acknowledge the completion of a requested action.
ADR_O[x..y]	Standard Wishbone address bus, used to select an address in a connected Wishbone slave device for writing to/reading from.
CYC_O	Cycle signal. When asserted, indicates the start of a valid Wishbone bus cycle. This signal remains asserted until the end of the bus cycle, where such a cycle can include multiple data transfers
ERR_I	Error input signal. Used to indicate the abnormal termination of a cycle.

RTY_I	Retry input signal. Informs the Wishbone Master interface that the Wishbone Slave is not ready to send or receive data at this time and the data exchange should be retried.
SEL_O[x..y]	Select output signal. Used to indicate where data is placed on the DAT_O line during a Write cycle and from where data is accessed on the DAT_I line during a Read cycle. Data ports have both a width and a granularity. For example a 32-bit wide data port with 8-bit granularity would allow data transfers of 8- 16- or 32-bits. The Select bits allow targeting of each of the bit ranges with 0 corresponding to the lowest range and incrementing from there.
STB_O	Strobe signal. When asserted, indicates the start of a valid Wishbone data transfer cycle.
WE_O	Write enable signal. Used to indicate whether the local bus is a Read or Write cycle. 0 = Read 1 = Write

Wishbone Slave Signals

Signal Name	Signal Description
ACK_O	Standard Wishbone device acknowledgement signal. When this signal goes high, the Wishbone Slave has finished execution of the requested action and the current bus cycle is terminated.
ADR_I[x..y]	Standard Wishbone address bus, used to select an internal register of the Wishbone slave device for writing to/reading from.
CYC_I	Cycle signal. When asserted, indicates the start of a valid Wishbone cycle
ERR_O	Error output signal. Used to indicate the abnormal termination of a cycle.
RTY_O	Retry output signal. Informs the Wishbone Master interface that the Wishbone Slave is not ready to send or receive data at this time and the data exchange should be retried.
SEL_I[x..y]	Select input signal. Used to indicate where data is placed on the DAT_O line during a Read cycle and from where data is accessed on the DAT_I line during a Write cycle. Data ports have both a width and a granularity. For example a 32-bit wide data port with 8-bit granularity would allow data transfers of 8- 16- or 32-bits. The Select bits allow targeting of each of the bit ranges with 0 corresponding to the lowest range and incrementing from there.

STB_I	Strobe signal. When asserted, indicates the start of a valid Wishbone data transfer cycle.
WE_I	Write enable signal. Used to indicate whether the current local bus cycle is a Read or Write cycle. 0 = Read 1 = Write

3.3 Pin naming conventions for wishbone signal pins

Altium Designer utilizes several prefixes when naming schematic symbol pins that are a part of the wishbone interface. These prefixes are intended to make the connection of Wishbone compatible devices easier and more intuitive. The following table provides a list of common prefixes used in Altium Designer, though not a part of the Wishbone Interface standard. Prefixes will be applied following the convention:

`<prefix>_<wishbone signal name>`

So for example, the prefix **ME** indicates the interface to the memory bus on the host processor, thus the pin name **ME_STB_o** would indicate the Strobe Signal output on the memory interface of a device such as TSK3000A.

Common Prefixes Used in Wishbone Interface Signal Pin Names

Interface Prefix	Prefix Description
ME	<i>Processor</i> interface to the memory bus of the host processor
me	<i>Peripheral</i> interface to the memory bus of the host processor
IO	<i>Processor</i> interface to the IO bus of the host processor
io	<i>Peripheral</i> interface to the IO bus of the host processor
WBM	Wishbone Master interface on devices having both Master and Slave interfaces (e.g. BT656)
WBS	Wishbone Slave Interface on devices having both Master and Slave interfaces (e.g. BT656)

Common Prefixes Used in Wishbone Interface Signal Pin Names (cont.)

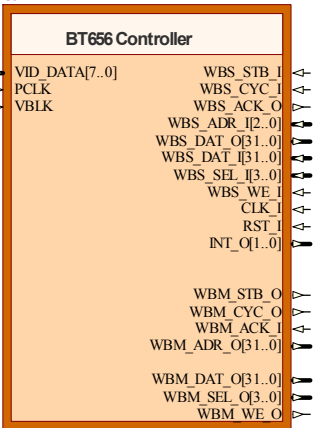
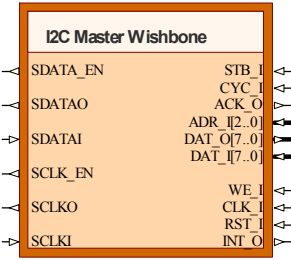
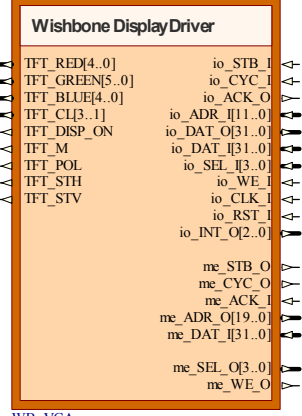
Interface Prefix	Prefix Description
m<number>_	Indicates a Wishbone Master interface to a Wishbone Interconnect or Wishbone Multi Master device. Wishbone Interconnect devices are restricted to a single master (typically the host processor), thus the Master interface name will always be prefixed by m0_. Wishbone Multi Master devices allow the master interfaces to be renamed thus m1_, m2_, m3_ might be renamed to MCU_, ASP_, VID_ in the device's configuration dialog.
s<number>_	Indicates a wishbone slave interface to a wishbone interconnect device. The <number> attribute uniquely identifies the various interfaces when connecting more than one slave peripheral to a given master. For example S0, S1, S2, etc. would all contain a complete wishbone interface signals for slave devices 1, 2, & 3 respectively.

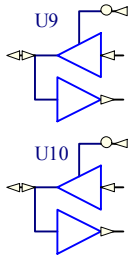
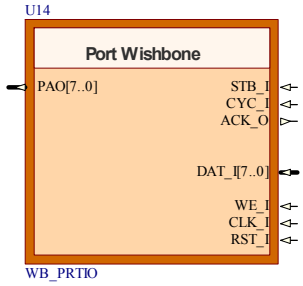
4 Placing the FPGA Peripherals

To complete the FPGA portion of our design, several peripherals are required to round out the overall system. As our application will capture video and display that video on a touch screen, it is easy to assume that we'll need at a minimum a video capture controller, a video display driver, and some memory both for the processor, as well as for the capture and display devices.

4.1 Exercise 3 – Placing the peripherals

In order to complete this task you will need to use the following components from their respective libraries:

Component	Library	Name In Library	Ref Des
U5  BT656	FPGA Peripherals.IntLib	BT656	U5
U7  I2CM_W	FPGA Peripherals.IntLib	I2CM_W	U7
U17  WB_VGA	FPGA Peripherals.IntLib	WB_VGA	U17

Component	Library	Name In Library	Ref Des
	FPGA Generic.IntLib	IOBUF	U9, U10
	FPGA Peripherals.IntLib	WB_PRTIO	U14

1. Place the BT656 Controller core `BT656` – this will be used to capture the digital video signal from the video decoder. Designate this component U5
2. Select and place the I2C Master Wishbone core `I2CM_W`. This will provide the intermediate I2C controller to control the video decoder. Designate this part U7.
3. Select and place the configurable Wishbone Display Driver `WB_VGA`. This will fetch the 16bpp-formatted data from external memory and display it on the TFT panel. Designate this part U17.
4. Right-click the Wishbone Display Driver `WB_VGA` and select Configure. Configure the component to match the figure below:



5. We will also require two I/O buffers for the I2C interface from the I2C Master Controller core to the SDA and SCL signals on the video decoder. Select the `FPGA Generic.IntLib` from the Libraries panel and place two IOBUF components. Designate them U9 and U10.
6. Though not a required component of our system, we will also make use of a Configurable Wishbone Port `WB_PRTIO` as a means to test our code. Designate this component U14.
7. Align all of your peripheral components along the left side of your processor as seen in Figure 6.

5 Bus Interconnectivity and Arbitration

Unlike off-the-shelf microprocessors which have their peripherals fixed at the time of manufacture, FPGA-based platforms have total flexibility. The choice in the number and type of peripherals is totally configurable and can be tailored to the specific needs of the application. This also adds a layer of complexity however, that until Altium Designer and the advent of interconnect architectures such as wishbone was difficult to manage.

Specifically, there exists the problem of how the peripherals interact with the processor when more than one peripheral needs access to a shared peripheral bus. This process historically was handled with glue logic to decode the address of the peripheral with which the processor was intended to communicate.

Further adding the complexity are devices that require direct memory access (DMA) and the management of multiple devices which again need share a single bus when interfacing to memory. Altium Designer addresses both of these issues with the Configurable Wishbone Interconnect, WB_INTERCON, and Configurable Wishbone Dual Master WB_DUALMASTER and Multi Master WB_MULTIMASTER components.

5.1 Wishbone Interconnect Component WB_INTERCON

The WB_INTERCON peripheral device provides a means of accessing one or more Wishbone-compliant slave devices over a single Wishbone interface. Connecting directly to either the External Memory or Peripheral I/O Interfaces of a processor, the device facilitates communication with physical memory devices or I/O peripherals, respectively.

The WB_INTERCON features:

- Complete configurability on the schematic sheet
- 1-to-n multiplexing (! Wishbone Master interface, multiple Wishbone Slave interfaces)
- Ability to control decoder address width
- Automatic hardware decoder generation
- Ability to define specific mapping into Processor address space
- 8-, 16-, and 32-bit Slave peripheral support
- Configurable addressing modes – allowing a slave device to be either byte or “word” addressed

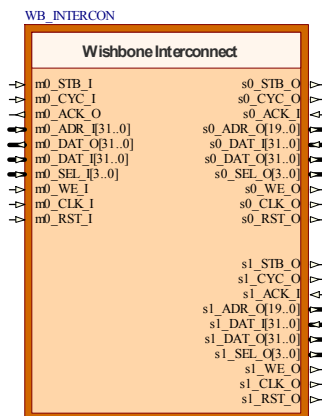


Figure 7. Wishbone Interconnect Component

5.2 Configuring the Wishbone Interconnect component

The Wishbone Interconnect component can be configured by right-clicking the component in schematic and selecting **Configure <RefDes> (WB_INTERCON)** or by launching the component properties dialog and hitting the configure button at the lower left.

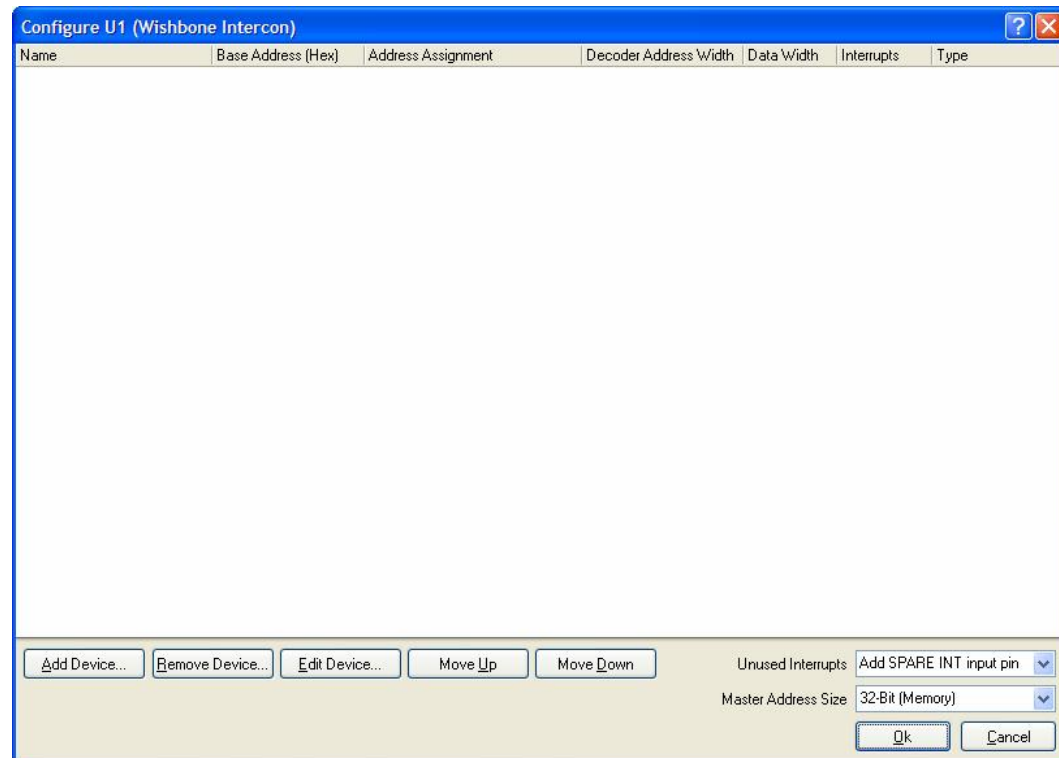


Figure 8. Wishbone Interconnect Component configuration dialog

In this dialog are options to **Add**, **Remove**, and **Edit** the banks of Wishbone Slave device interfaces, as well as options to **Move Up** and **Move Down**, physically reordering the location of the banks with respect to the schematic symbol.

The **Unused Interrupts** drop-down is used to control the behavior of any unused interrupts on the host processor. The available options are:

- **Add SPARE INT input pin** – use this option to make the unused interrupt signals available as an additional input pin. Interrupts from additional circuitry in the design can be wired into this pin.
- **Connect to GND** – use this option to internally connect all unused interrupts to GND. The interrupt output signal pattern that is sent to the Wishbone Master device will contain '0' for each of these unused interrupts.
- **No Interrupt output pin** – use this option to effectively disable marshalling of interrupts to the processor. Any configured interrupt input pins for slave devices will be removed from the symbol, as well as the spare interrupts pin (where applicable) and the output pin to the Wishbone Master.

The **Master Address Size** drop-down is used to specify the address size depending on the type of slave peripheral(s) being connected. The **32-Bit(Memory)** option is used when connecting to slave memory devices. If using the device to connect to slave peripheral I/O devices, ensure that this option is set to **24-Bit(Peripheral I/O)**.

Again, the order in which the devices appear in this dialog will determine the order in which the Wishbone Interface pins will appear on the schematic symbol, from top to bottom. Using **Move Up** and **Move Down** in the **Configure U1 (Wishbone Intercon)** dialog will reorder the groups of signals as they appear in schematic.

The Add Device option is used to add a new wishbone slave device. This will launch the Device Properties dialog that appears in figure 9.

Figure 9. Adding a device to the wishbone interconnect component

The following table describes the various properties that can be defined for a wishbone slave interface.

Interface Property	Description
Name	Then name is a user-definable field and should meaningful when listed alongside other connected devices.
Type	Used to specify the type of Wishbone Slave Device is being connected.

Interface Property	Description
Address Bus Mode	Select either Word or Byte addressing. In byte addressing mode, all of the lower address lines are passed to the slave, regardless of the resolution of its data bus. The slave device will handle “byte-to-word” management. In word addressing mode, the mapping of the address lines are passed to the slave device’s data bus width: • 32-bit wide devices – the two lowest address bits are not connected to the slave device. ADR_I(2) from the master is mapped to ADR_O(0) of the slave, providing sequential word addresses (or addresses at every 4 bytes). Registers/address locations in such devices can be read and written using the LW and SW 32-bit load/store instructions • 16-bit wide devices – the lowest address bit is not connected to the slave device. ADR_I(1) from the master is mapped to ADR_O(0) of the slave, providing sequential half-word addresses (or addresses at every 2 bytes). Registers/address locations in such devices can be read and written using the LHU and SH 16-bit load/store instructions • 8-bit wide devices – all address bits are connected through to the slave device. ADR_I(0) from the master is mapped to ADR_O(0) of the slave, providing sequential byte addresses. This is identical to byte addressing. Registers/address locations in such devices can be read and written using the LBU and SB 8-bit load/store instructions.
Address Base	Used to specify a decoder base address for a Slave device. A portion of this address – specified by the Decode Addressing value – will be compared against the corresponding bits of the incoming m0_ADR_I signal to determine whether the slave is being addressed or by the processor or not.
Decode Addressing	Defines the decoder address width. This value determines the number of upper address bits on the m0_ADR_I line that are decoded to select the correct slave device. There in turn this value also determines the number of slave devices that can be connected to the interconnect device.

Interface Property	Description
Address Bus Width	Used to specify the number of address bits required to drive the connected slave device. For slave memory devices – which are connected via the Memory Controller device – you need to set the address bus to the same width as the ADR_I line for the Memory Controller. The Memory Controller will automatically size its ADR_I line according to the size of the physical memory it is connecting to. For slave peripheral devices, you need to set the address bus to the same width as the ADR_I line for the peripheral.
Data Bus Width	Used to specify the resolution of the data bus for the slave device. 8-bit, 16-bit and 32-bit data bus widths are supported.
Used Interrupts	Used to select which interrupt(s) to use for the slave device. For each interrupt line, you can also define its type – whether it is level-sensitive or edge-triggered – as well as its polarity. Interrupts generated by Altium Designer Wishbone peripheral devices have positive polarity and are level-sensitive, and this is therefore the default setting. You can also specify the required interrupt pins to be used by directly typing within the available text field – in the <i>Device Properties</i> dialog. Any defined interrupts will appear as part of the overall 32-bit interrupt input bus sent to the Wishbone Master (e.g. a 32-bit host processor).
Graphical Attributes	Alters the amount of blank space inserted after a bank of pins. An increase of +1 unit = +10 schematic grid units or +100 mil spacing (1:1)

5.3 Exercise 4 – Connecting our Peripherals

Connecting the host processor to the various peripherals we've placed is accomplished using the Configurable Wishbone Interconnect component. In this exercise we will see how to configure this component for our design.

1. From the libraries panel, select the `FPGA Peripherals.IntLib` and place the Configurable Wishbone Interconnect component `WB_INTERCON` between the left side of the host processor and the right side of the peripherals already placed. Designate this component U1.
2. Notice that currently this device has only one set of Wishbone signal pins, prefixed `m0`. The `m0` prefix indicates the Master Interface and should be facing the left side of the host processor.
3. To configure this device for the slave devices, right-click U1 and select **Configure U1 (WB_INTERCON)...**
4. Referencing Figure 10, configure the following Slave Interfaces.

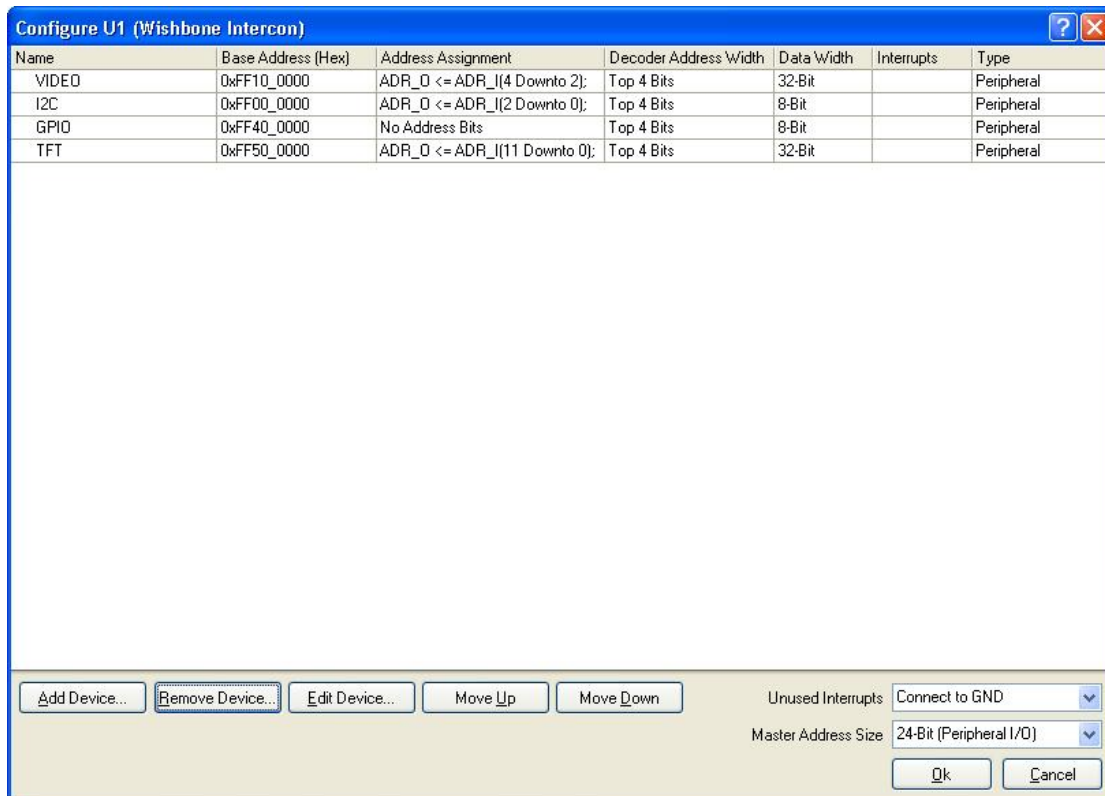


Figure 10. Wishbone interconnect component interfaces

Note: The video capture interface will require **Word** addressing while all other interfaces will use **Byte** addressing.

5. Set the **Unused Interrupts** dropdown to **Connect to GND** and the **Master Address Size** dropdown to **24-Bit(Peripheral I/O)**.

5.4 Managing memory access

Like their traditional 'hard'-processor counterparts; systems on FPGAs require memory resources. Whether these resources exist on-chip or off-chip depends on the needs of the system and the device selected. Generally, on-chip memory is faster but consumes device resources. Off-chip memory on the other hand, though generally slower, has potentially greater storage capacity.

For our application the bulk of our memory resources will exist off-chip as SRAM on the Spartan 3 Daughtercard. The memory controllers however will exist on-chip, as will the arbiters that will manage memory access.

5.5 Arbitration of multiple masters sharing a single slave

In our system, we are again faced with a single bus to connect multiple soft devices, though in the case of memory the challenge is somewhat different as rather than managing a single master to multiple slaves, we are instead made to manage multiple masters sharing a single slave interface. The single [slave] interface exists on the memory controller side and arbitration must occur to ensure that there is never more than one master device accessing memory at a time. Likewise, there are issues of priority that arise.

In our design, one block of memory will be shared by the Capture Controller and the Host Processor. Another will be shared by the Display Driver and the Host Processor. The

processor will effectively access the video data from the Capture Memory and then write that data to the Display Memory, zoomed and rotated per the user's input.

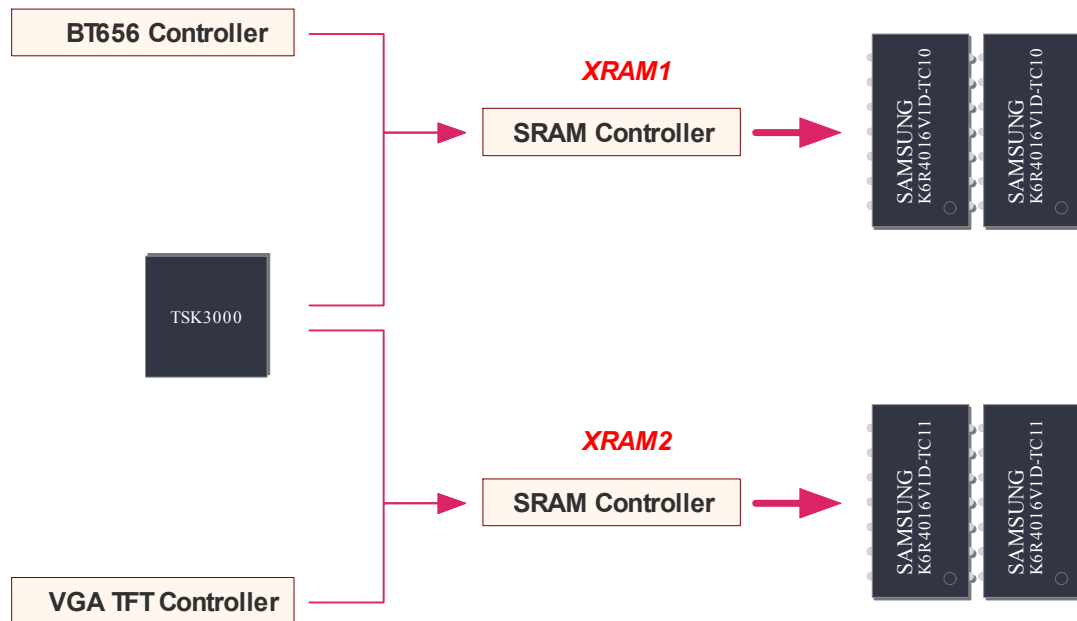


Figure 11. Block diagram showing the layout of the memory system

This means that for each of the two blocks of memory, we will require an *arbiter* to control memory access. This will ensure that both the host processor and the peripheral (whether the Capture Controller or the Display Driver) are not attempting to access the same memory at the same time.

5.6 Wishbone Dual- and Multi- Master devices

Altium Designer includes two types of arbiters, the Wishbone Dual Master (for managing two Masters' access to a single Slave) and the more extensible Wishbone Multi Master component (for managing up to 8 Masters' access to a single Slave device).

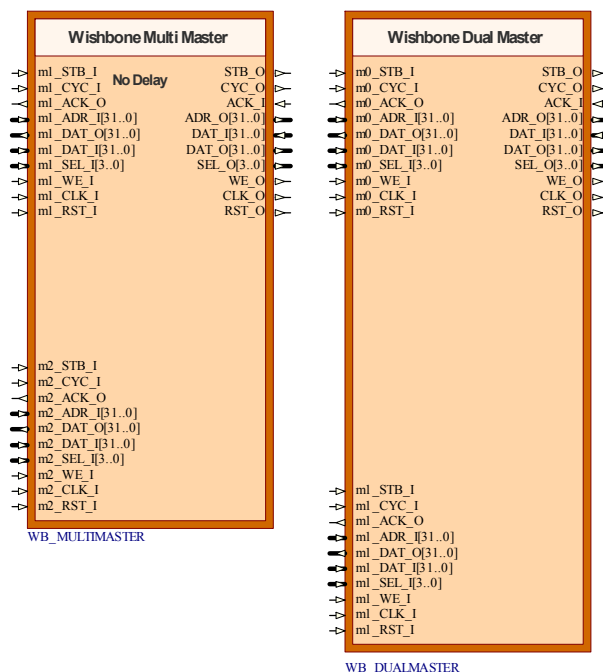


Figure 12. Wishbone dual- and multi-master components

As we will be making changes later that will require the addition of another Master accessing each of these two memories, we will utilize the Wishbone Multi Master components in our design. The Wishbone Multi Master features:

- Support for up to 8 Wishbone Masters
- Ability to control address and data bus widths
- Ability to specify Master accessing mode:
 - Round-Robin – giving masters access to the slave in sequence (top to bottom)
 - Priority – allowing you to define an order of priority for access to the slave
- Ability to grant one Master instant access to the bus when in 'idle' state
- Supports passing of interrupts from a connected Wishbone Interconnect, through to all connected 32-bit processors (when used to share slave peripheral I/O devices).

5.7 Configuring the wishbone multi- master component

The Wishbone Interconnect component can be configured by right-clicking the component in schematic and selecting **Configure <RefDes> (WB_MULTIMASTER)** or by launching the component properties dialog and hitting the **Configure** button at the lower left.

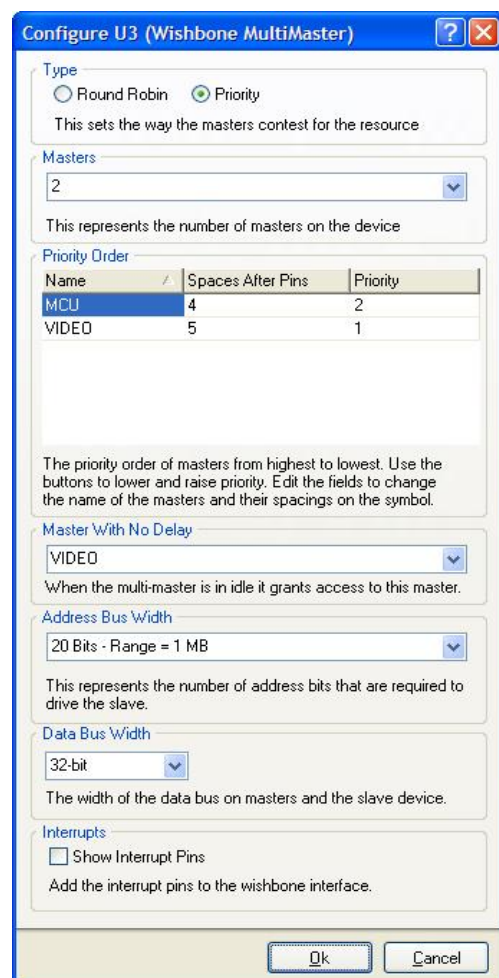


Figure 13. Wishbone multi- master configuration dialog

In this dialog are options to **Add**, **Remove**, and **Edit** the banks of Wishbone Slave device interfaces, as well as options to **Move Up** and **Move Down**, physically reordering the location of the banks with respect to the schematic symbol. The following table lists the various options available for configuring masters' access to a slave device.

Option	Description
Type	Enables you to specify the way in which the Wishbone Masters contest for the slave resource. The following options are available: • Round Robin - masters access the slave in sequence, from first (top) to last (bottom). • Priority – masters access the slave in the specified order of priority.
Masters	Allows you to specify how many master interfaces the device will have (up to 8 masters are supported)
Round Robin / Priority Order	Displays the Type and number of masters specified. If Type is set to Round Robin, only the Name and Spaces After Pins columns will be displayed. The shared resource will be accessed by each master in turn, from the top of the list down. If Type is set to Priority, a third Priority column will be displayed allowing you to specify an explicit priority (1 being highest priority). Use the Name field to change the prefix for each master. Use the Spaces After Pins field to determine the amount of blank space that is inserted after the bank of pins for that master interface.
Masters With No Delay	Allows you to specify one master to be granted instant access to the bus when the WB_MULTIMASTER is 'idle'. This reduces latency as the nominated master experiences no delay in acquiring access. Typically, when operating in Priority mode, the master specified here will be the one assigned highest priority. The interface assigned to be the master with no delay is distinguished on the schematic symbol by insertion of the text "No Delay".

Option	Description
Address Bus Width	Allows you to specify the number of address bits required to drive the connected slave device. The width chosen is applied to all interfaces of the WB_MULTIMASTER. When connecting to a single slave memory device you need to set the address bus to the same width as the ADR_I line for the Memory Controller. The Memory Controller will automatically size its ADR_I line according to the size of the physical memory it is connecting to. A Wishbone Interconnect must then be used between the Multi-Master and the processor's External Memory Interface, to handle the address line mapping. When connecting to a bank of physical memory devices through a Wishbone Interconnect, the address bus must be set to 32 Bits - Range = 4GB, which matches the ADR_I line of the Interconnect's master interface. When connecting to a single slave peripheral device, you need to set the Multi-Master's address bus to the same width as the ADR_I line for the peripheral. A Wishbone Interconnect must then be used between the Multi-Master and the processor's Peripheral I/O Interface, to handle the address line mapping. When connecting to a bank of peripheral devices through a Wishbone Interconnect, the address bus must be set to 24 Bits - Range = 16MB, which matches the ADR_I line of the <i>Interconnect's</i> master interface.
Data Bus Width	Allows you to specify the resolution of the data bus for the slave device being connected. 8-bit, 16-bit and 32-bit data bus widths are supported. The width chosen is applied to all interfaces of the WB_MULTIMASTER.
Interrupts	Enable the Show Interrupt Pins option in this region of the dialog to add the INT_O[31..0] and INT_I[31..0] pins to the master and slave interfaces respectively. The interrupt pins would be added when the Multi-Master device is used to connect multiple 32-bit processors to a bank of peripheral I/O devices, via a Wishbone Interconnect. This allows interrupts generated by those peripherals to be passed from the Interconnect through to the processors.

5.8 Exercise 5 – Defining our Multi Master Components

In this exercise we will place and configure the Wishbone Multi Master components required by our design. These components will function as arbiters to manage access to the external memory used by our system.

1. From the libraries panel, select the `FPGA_Peripherals.IntLib` and place 2 Wishbone Multi Master components `WB_MULTIMASTER` along the right side of the host processor. Designate these U3 and U12.
2. Right click U3 and select `Configure U3 (WB_MULTIMASTER)` to launch the configuration dialog. We will configure this device for both the host processor as well as the BT656 Video Capture Controller.
3. Configure U3 and U12 as seen in the figure below.

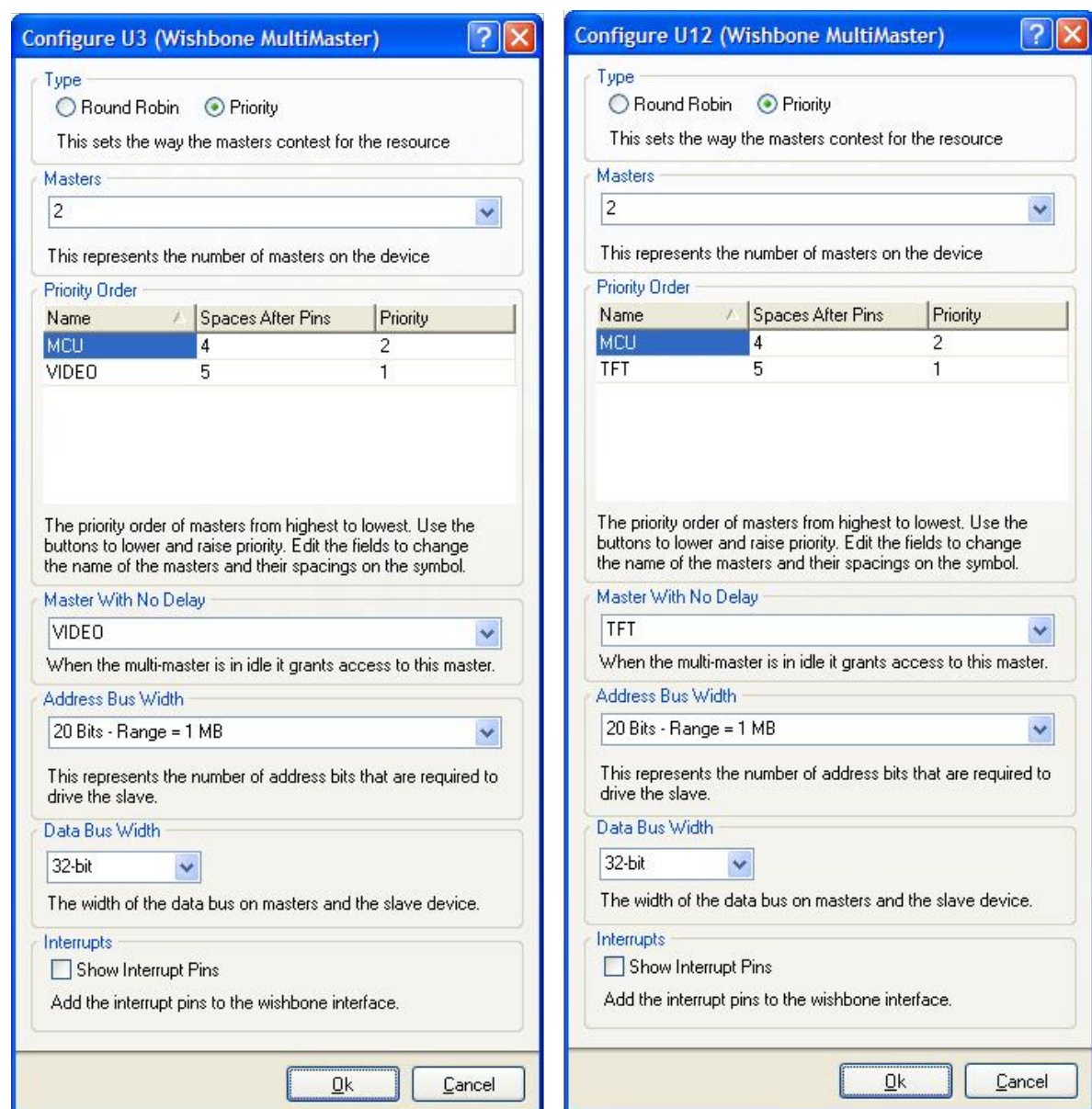


Figure 14. U3 and U12 Wishbone MultiMaster configuration dialogs

6 Interfacing to External Memory

Altium Designer includes a Wishbone Compatible Configurable Memory Controller `WB_MEM_CTRL` that, depending on its configuration, provides a simple interface between a 32-bit processor and either single data rate Synchronous DRAM, Asynchronous Static RAM, 32-bit wide Block RAM (single or dual port), or parallel Flash memory.

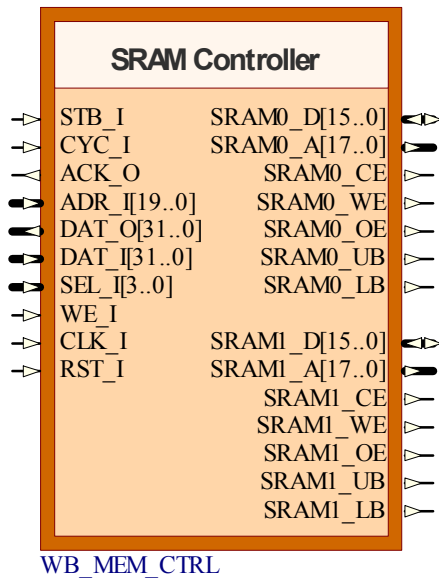


Figure 15. Wishbone Configurable Memory Controller

The Memory Controller (Figure 15) features:

- Simplified connection to processor's External Memory interface via a Wishbone Interconnect
- Configurable as either SRAM, SDRAM, or BRAM Controller
 - SDRAM Controller – interfaces to 8-, 16, or 32-bit wide SDR SDRAM
 - SRAM Controller – interfaces to 8-, 16, or 32-bit wide asynchronous SRAM
 - BRAM Controller – interfaces to 32-bit wide single or dual port Block RAM
 - Flash Controller – interfaces to 8-, 16, or 32-bit wide parallel Flash memory
- Automatic sizing of `ADR_I` input bus, based on specified physical memory size
- Wishbone-compliant interface

6.1 Configuring the SRAM Controller

The SRAM Controller component can be configured by right-clicking the component in schematic and selecting **Configure <RefDes> (WB_MEM_CTRL)** or by launching the component properties dialog and hitting the **Configure** button at the lower left.

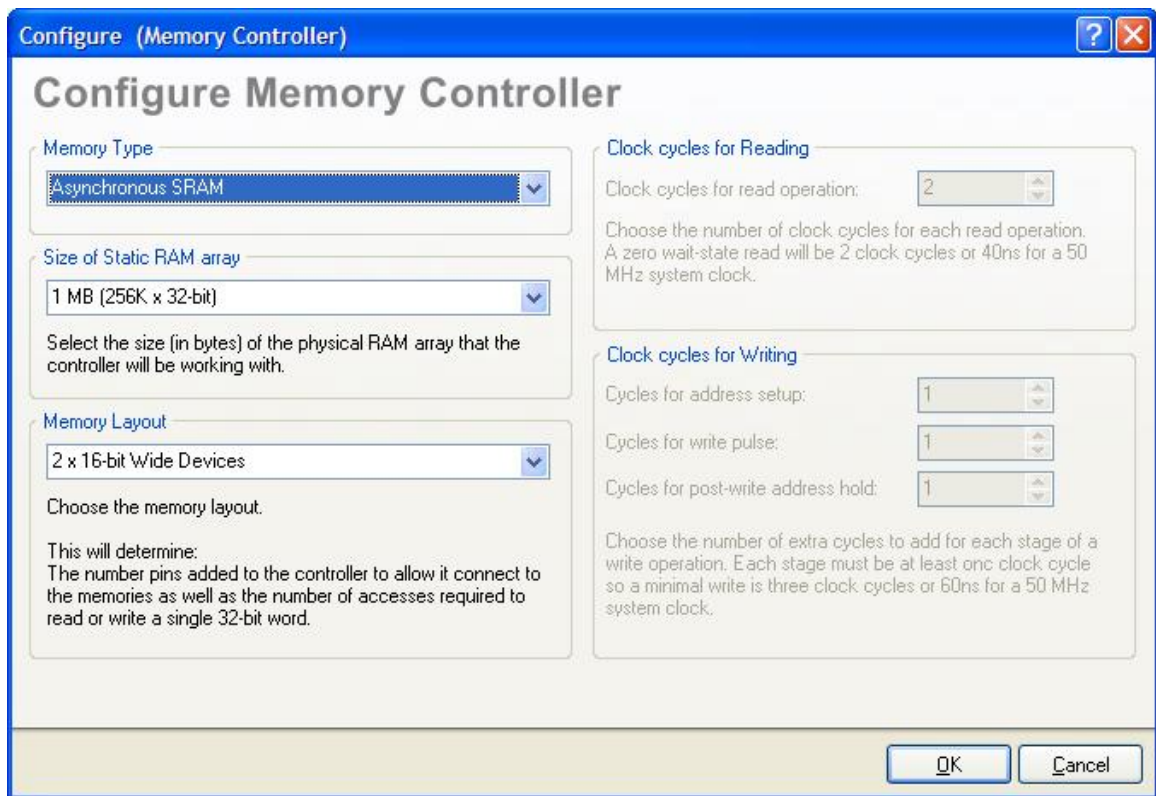


Figure 16. Wishbone Configurable Memory Controller configuration dialog

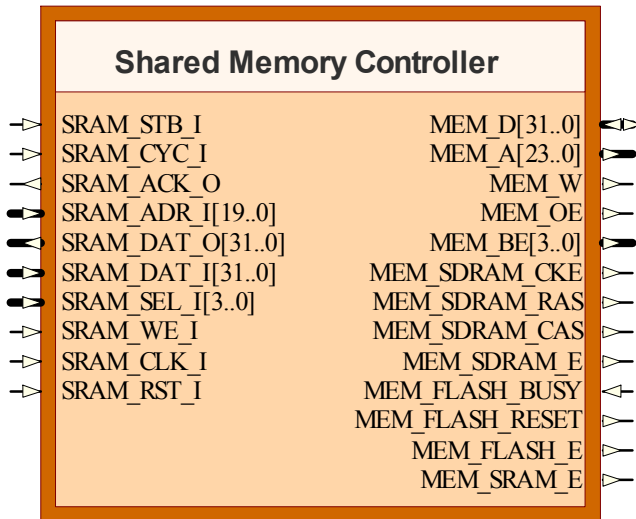
In this dialog are options to configure the memory controller for use with a variety of different memory sizes and types. The following table provides a description of the various options available when configuring the memory controller.

Option	Description
Memory Type	Specifies the type of memory that you wish to interface to. Choose from either: • Synchronous DRAM • Asynchronous SRAM • Block RAM • Parallel FLASH The configuration options presented in the dialog will change with your selection.
Size Of <Memory Type>	Specifies the size of the physical RAM that you are interfacing to. The width of the interface address bus(es) and also the ADR_I input line will update accordingly upon leaving the dialog. This dialog will update dynamically as the Memory Type changes.

Option	Description
Size Of <Memory Type>	Specifies the size of the physical RAM that you are interfacing to. The width of the interface address bus(es) and also the ADR_I input line will update accordingly upon leaving the dialog. This dialog will update dynamically as the Memory Type changes.
Memory Layout	Used to select the layout for the physical memory. The schematic symbol for the placed Memory Controller will automatically be updated to reflect your selection, upon leaving the dialog. In addition to determining the interface pinout for connection to the physical memory device(s), the memory layout also determines the number of accesses required to read or write a single 32-bit word. In the case of BRAM, this region of the dialog is non-editable and reflects the layout of physical BRAM that can be connected to the Controller
Clock cycles for Reading	Only editable when using Parallel Flash, this option allows you to specify the number of clock cycles (of CLK_I) required to perform a read operation.
Clock cycles for Writing	Only editable when using Parallel Flash, this option allows you to specify the number of clock cycles (of CLK_I) required to perform a write operation.
Memory Settings	Only visible when using SDRAM, this option allows you to specify the SDRAM's clock frequency in MHZ. The frequency of the signal arriving at the controller's SDRAM_CLK input must match this clock frequency. The Keep Current Row Option allows you to keep the current row open – essentially disabling the SDRAM device's auto-precharge feature which otherwise causes a precharge of the addressed bank/row upon completion of the current Read or Write.
Timer Settings	Only visible when using SDRAM, these options allow you to specify the Write Recovery Time (tWR), Auto Refresh period (tRFC), Active to Read or Write delay (tRCD), Precharge command period (tRP), and CAS latency (tCAS). Each of these values is specified in terms of the number of cycles of the SDRAM_CLK signal. The dropdown allows you to define the synchronization scheme to be used when interfacing to SDRAM. Ensure that the frequency of the signal wired to the Controller's SDRAM_CLK input is in accordance with the synchronization setting you have chosen.

6.2 The Shared Memory Controller

Altium Designer also includes a Wishbone Compatible Shared Memory Controller core `WB_SHARED_MEM_CTRL` that, depending on its configuration, provides a simple interface between a 32-bit processor and memories on a shared bus.



[WB_SHARED_MEM_CTRL](#)

The Controller provides access to, and use of, the following three different types of memory, each of which is multiplexed for access over shared data and address busses:

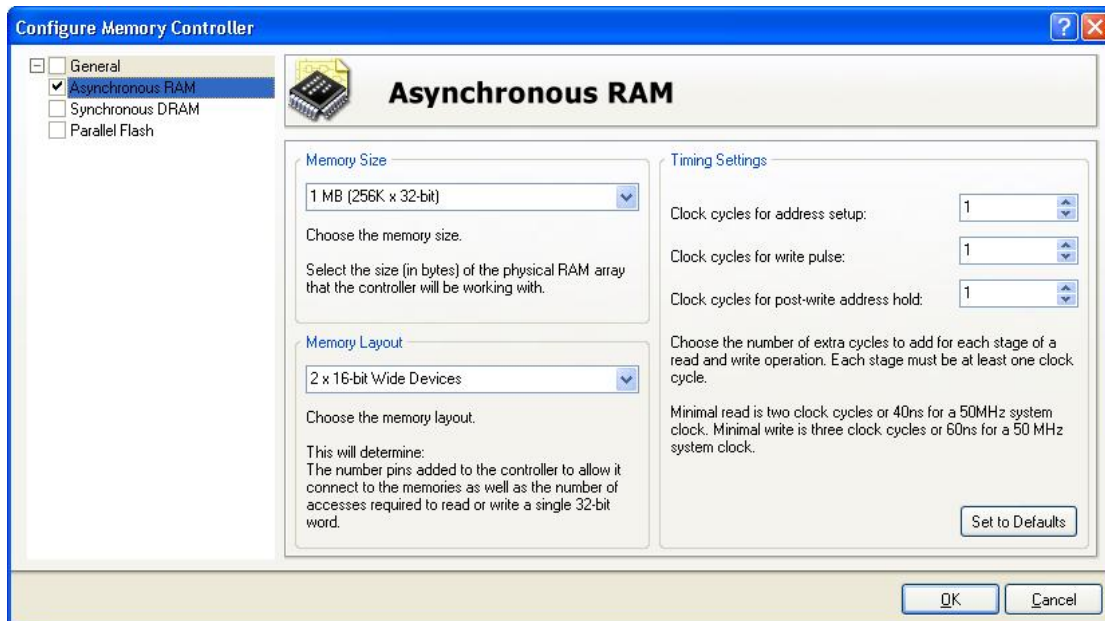
- Asynchronous Static RAM
- Single data rate Synchronous DRAM
- Parallel Flash memory

The Controller handles all multiplexing for you, negating the need for custom demultiplexing logic.

Note: The WB_SHARED_MEM_CTRL is primarily designed to be used with the common-bus memories located on Altium's 3-connector daughter boards, such as the Xilinx Spartan-3 Daughter Board DB30. Provided the same pinout is used, the Controller could be used to interface to other memories of the types supported, and which are accessed using a shared bus architecture.

6.3 Configuring the Shared Memory Controller

The SRAM Controller component can be configured by right-clicking the component in schematic and selecting **Configure <RefDes> (WB_SHARED_MEM_CTRL)** or by launching the component properties dialog and hitting the **Configure** button at the lower left.



In this dialog are options to configure the memory controller for use with a variety of different memory sizes and types. For a complete list of configuration options, see the reference [CR0176 WB_SHARED_MEM_CTRL Configurable Shared Memory Controller.pdf](#) found under the Help menu our available from the Altium website. For our design we will be configuring the device as Asynchronous RAM and the following table provides a description of the various options available when configuring the memory controller.

Option	Description
Memory Size	Use this region of the page to specify the size of the physical memory that you are interfacing to. For example, the common-bus SRAM on a 3-connector daughter board is provided by two 4Mbit, high-speed CMOS SRAM devices. Each device is organized as 256K x 16 bits – combined together to give 256K x 32-bit storage (1MByte). The width of the SRAM_ADR_I input line will automatically change according to memory size specified.
Memory Layout	Use the drop-down available in this region of the page to select the layout for the physical memory. In addition to determining the interface pinout for connection to the physical memory device(s), the memory layout also determines the number of accesses required to Read or Write a single 32-bit word.
Timing Settings	This region of the page enables you to specify additional clock cycles (cycles of SRAM_CLK_I) to be added for each stage of a Read and Write operation. Each stage must be at least one clock cycle. The minimum number of clock cycles for each operation are: • Read – two clock cycles. If the system clock (SRAM_CLK_I) is 50MHz, this equates to 40ns.

	• Write – three clock cycles. With a system clock (SRAM_CLK_I) of 50MHz, this equates to 60ns. The following default timing settings are used: • Clock cycles for address setup – 1 cycle • Clock cycles for write pulse – 1 cycle • Clock cycles for post-write address hold – 1 cycle.
--	--

6.4 Exercise 6 – Placing and configuring memory controllers

In this exercise we will place and configure the Configurable Wishbone Memory Controller `WB_MEM_CTRL` and the Wishbone Shared Memory Controller `WB_SHARED_MEM_CTRL`. These controllers will serve as the FPGA-side interface to 2MB of Asynchronous SRAM located on the Spartan 3 FPGA Daughterboard.

1. From the libraries panel, select the `FPGA Peripherals.IntLib` and place the Configurable Memory Controller components `WB_MEM_CTRL` along the right side of the Wishbone Multi Master component. Designate this component U4.
2. Right click U4 and select **Configure U4 (WB_MEM_CTRL)...** to launch the configuration dialog. Configure U4 as it appears in Figure 17.

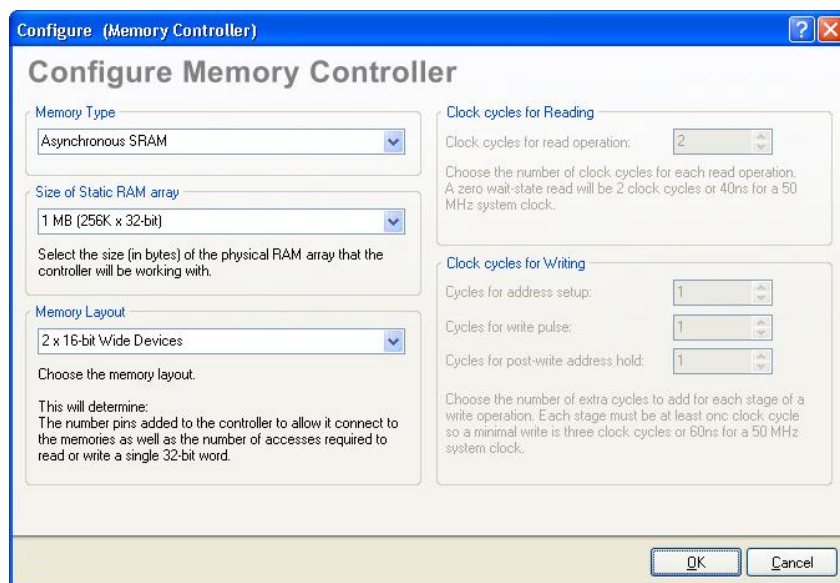


Figure 17. Configuring the wishbone memory controller U4

3. Return to the `FPGA Peripherals.IntLib` and place the Shared Memory Controller components `WB_SHARED_MEM_CTRL` along the right side of the Wishbone Multi Master component. Designate this component U13.
4. Right click U13 and select **Configure U13 (WB_SHARED_MEM_CTRL)...** to launch the configuration dialog. Configure U13 as it appears in Figure 18.

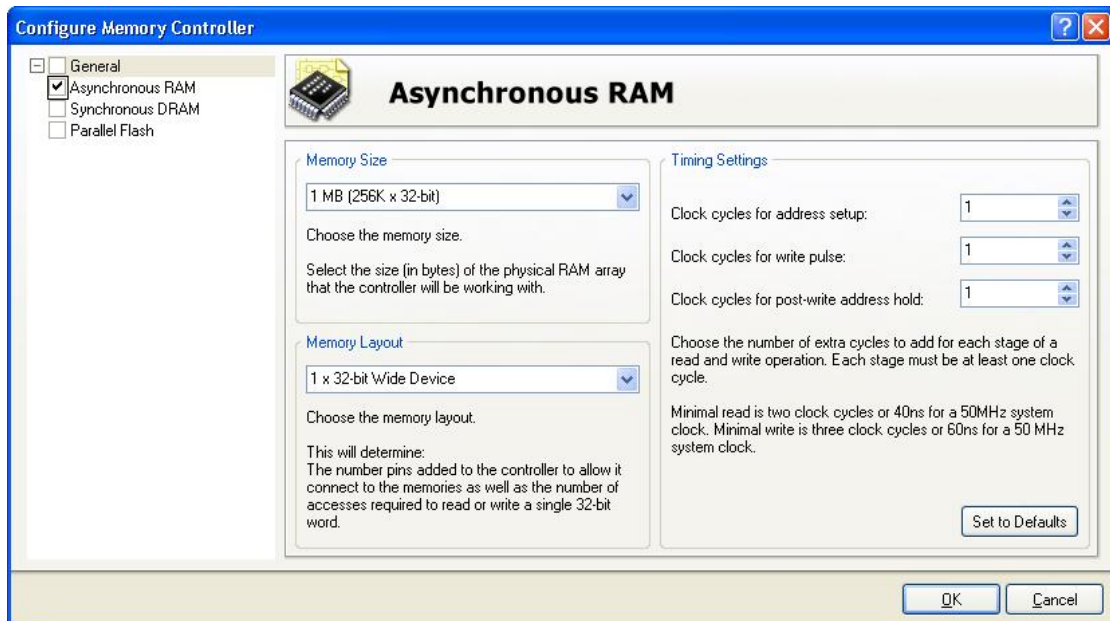


Figure 18. Configuring the wishbone shared memory controller U13

6.5 Exercise 7 – Multiple memories as slaves

Our host processor will require a second Wishbone Interconnect component as each of the two arbiters appears as slaves to the processor's memory interface. In this exercise we will add a second Configurable Wishbone Interconnect component and configure it for the two memories in our design.

1. From the libraries panel, select the `FPGA Peripherals.IntLib` and place a Configurable Wishbone Interconnect component `WB_INTERCON` to the right side of the host processor. Designate this component U8.
2. Right click the component and select `Configure U8 (WB_INTERCON)...` to launch the component's configuration dialog.
3. In the configuration dialog, configure this component as it appears in Figure 19.

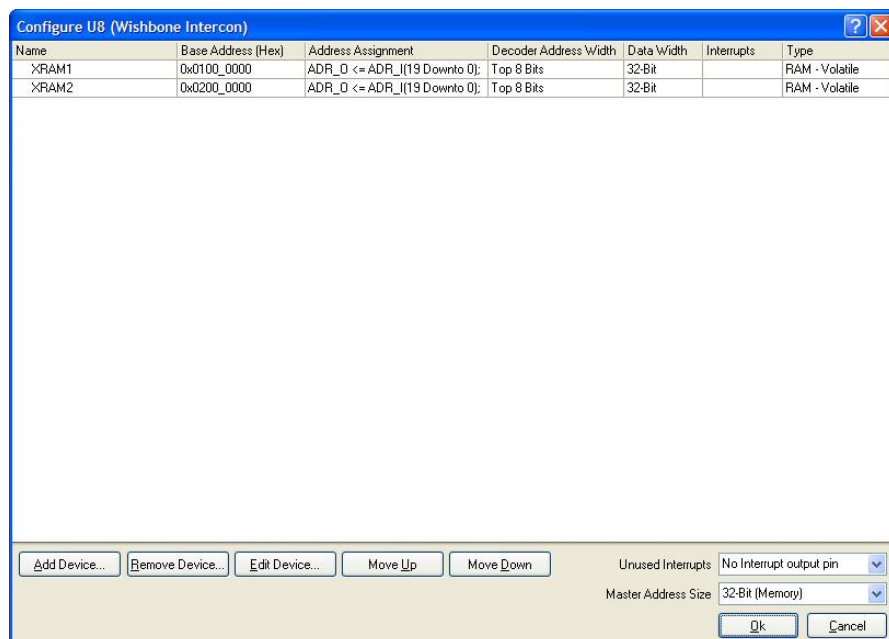


Figure 19. Configuring the wishbone interconnect to support multiple memory slaves

6.6 Exercise 8 – Placing the Port Plug In Components

Each of the peripherals that we have placed thus far will connect to the corresponding hardware on the NanoBoard using a Port Plug In component located in one of the various Port Plug In libraries detailed in our discussions on Day 1. In this exercise we will place the required Port Plug In components and arrange them such that we might simplify the connectivity.

1. Locate the FPGA PB01 Port-Plugin.IntLib in the **Libraries** panel and place the following components arranged along the left side of the FPGA peripherals as seen in Figure 20.
 - VIDEO_INPUT
 - VIDEO_INPUT_CTRL
2. Locate the FPGA NB2DSK01 Port-Plugin.IntLib in the **Libraries** panel and place the following components arranged along the left side of the FPGA peripherals as seen in Figure 20.
 - CLOCK_BOARD
 - TEST_BUTTON
 - LED
 - TFT_LCD
 - NEXUS_JTAG_CONNECTOR
3. Locate the FPGA DB Common Port-Plugin.IntLib in the **Libraries** panel and place the following components arranged along the left side of the FPGA peripherals as seen in Figure 20.
 - SRAM_DAUGHTER0
 - SRAM_DAUGHTER1
 - SHARED_SRAM_DAUGHTER
4. Locate the FPGA Generic.IntLib in the Libraries panel and place the following components.
 - NEXUS_JTAG_PORT
 - INV
5. Designate the Inverter U16.

No reference designators will be required for any parts other than the inverter as the remaining parts are not components of the FPGA but rather ports referencing components external to the FPGA project. The fact that these have been collected into a library and can be placed as components is intended only to simplify the process of developing FPGAs using the Desktop NanoBoard.

The schematic in Figure 20 displays how to arrange the components to ensure ease in wiring. Notice the way in which each of the Wishbone Interfaces can be aligned to simplify this process.

Figure 20. Schematic including the required port plug-in components

6.7 Exercise 10 – Wiring Up Your FPGA Design

In this exercise we will wrap up the capture portion of our design by wiring up our schematic. As in Day 1, these wires represent physical connections inside the FPGA whether between blocks of FPGA IP or to pins of the FPGA device (as is the case of the NanoBoard Port Plug In components).

With the assistance of your instructor, wire the design as it appears in Figure 21. Snippets have been added to the Snippets panel to assist you in making this process faster. Be sure to include bus joiners and bus power ports as required.

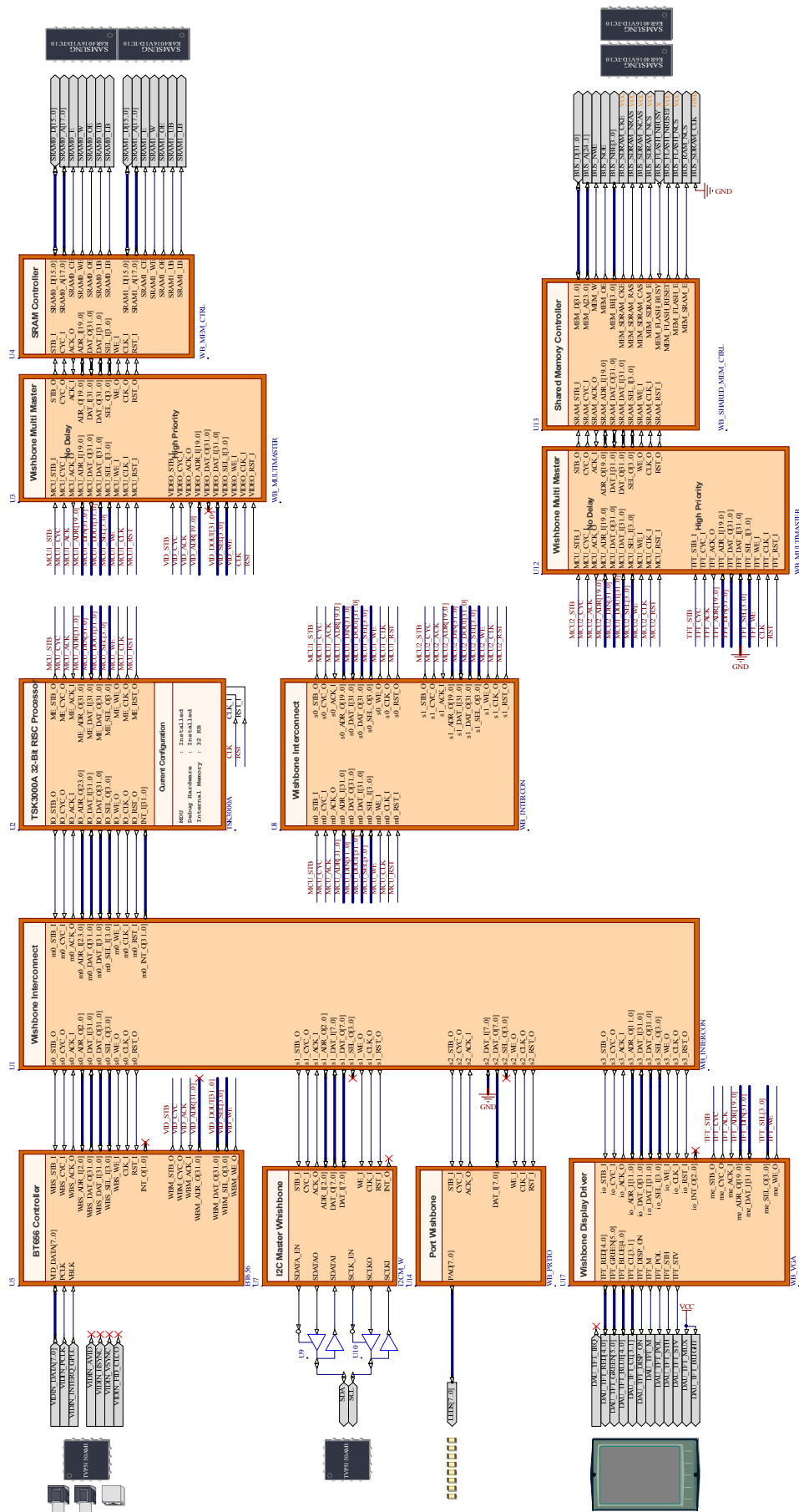


Figure 21. Completed schematic with the required wiring

7 Configuring Processor Memory

At this stage we have largely completed the FPGA portion of our design however one key step remains and that is the configuration of our processor memory from within the processor.

In addition to the memory that is managed by each of the interconnect components within our design; you can also centrally manage memory from the processor's memory and peripheral configuration dialog boxes. These are accessed by right-clicking the processor and selecting **Configure Processor Memory...** and **Configure Processor Peripheral...** respectively. Alternatively, both dialogs are accessible from within the component properties dialog for these components.

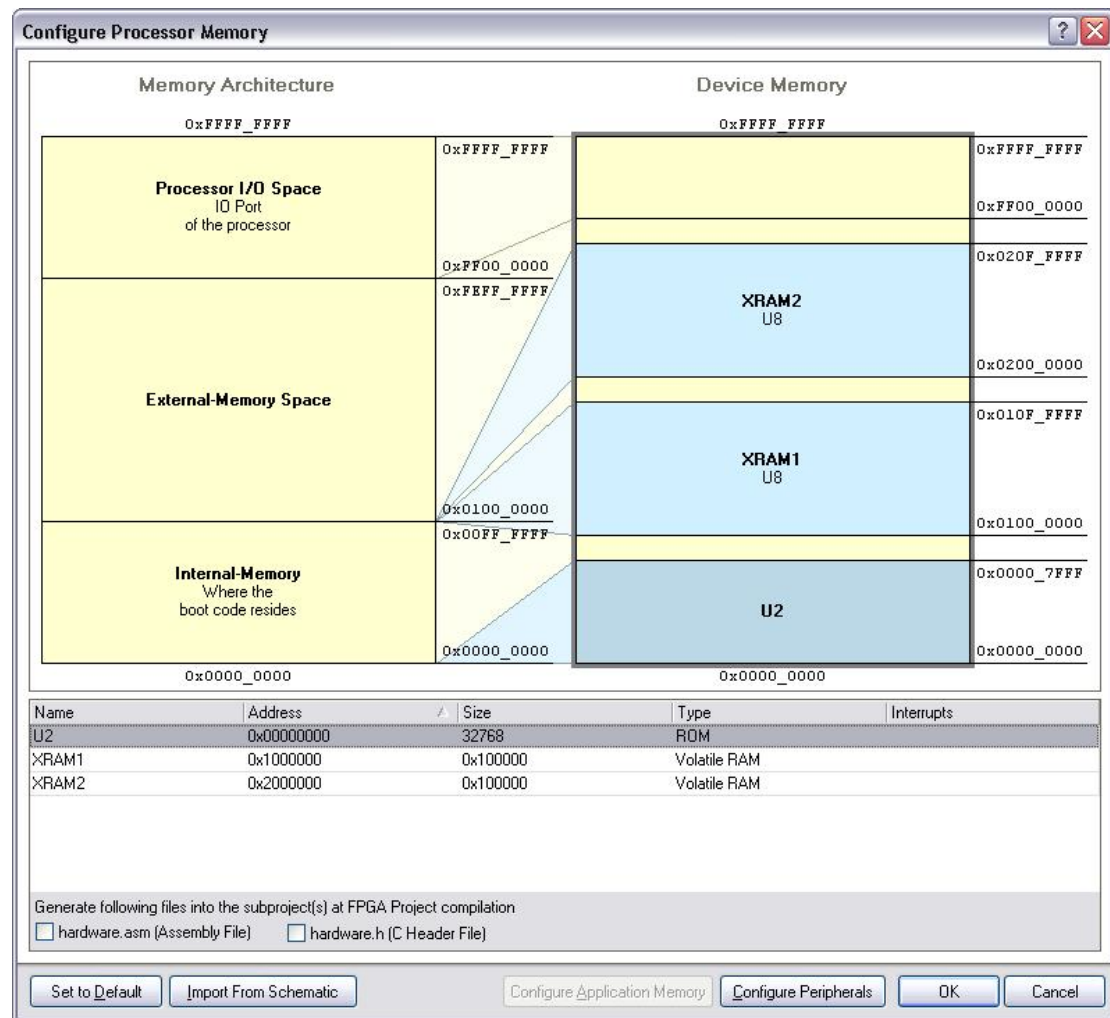


Figure 22. Processor memory configuration dialog

Below is a table that describes the various options within this dialog.

Option	Description
hardware.asm (Assembly File)	Used to pass the memory and peripheral definition information to the Embedded Software project in the form of a Header File. A header file essentially provides keyword substitution. The Embedded Software designer can simply enter a definition into the code, which, through the header file, will be substituted with the required information at compile time. This option produces a Header file in Assembly language.
hardware.h (C Header File)	Used to pass the memory and peripheral definition information to the Embedded Software project in the form of a Header File. This option produced a Header file as a C language file.
Set to Default	Used to Reset the memory configuration to its default settings. This option would only be used when the current memory configuration is out of date and needs to be re-done, or when the design does not contain a special or explicit configuration. Use caution as this option does not provide a Warning dialog and can only be undone by hitting Cancel prior to closing the dialog or closing and reopening the schematic.
Import From Schematic	Used when physical memory devices are connected to the processor through a Wishbone Interconnect. This is the simplest way to make these devices 'known' to the processor as it automatically maps the devices into the processor's address space per the settings defined in the Wishbone Interconnect component's configuration dialog (as seen with the XRAM slave interfaces which had come up predefined in Exercise 10).
Configure Application Memory	Used to define a memory configuration specific to a particular application.
Configure Peripherals	Launches the Configure Peripherals dialog used to specify the peripherals as they are 'seen' by the processor through its Processor I/O space. If using Wishbone compliant peripherals accessed through the Wishbone interconnect, this dialog may be populated automatically using the Import From Schematic option.

7.1 Dividing the processor memory

Though we have already defined the size of the Internal Processor Memory when we'd previously configured the device (this was 32K Bytes or 8K x 32-bit Words); Altium Designer affords us the ability to further carve out specific memories to appear in a particular way to the host processor. So for example, though we have 32 Kbytes of memory, we will actually change the way the processor 'sees' this memory, carving it into 2, 16 Kbyte chunks.

For our application, we will also need to partition the processor's internal memory, specifying half of the internal memory (16 Kbytes) as ROM and the other half as RAM. To do this, we need first to right-click in the **Device Memory** section of the **Configure Processor Memory** dialog and remove the large, 32 Kbytes section labeled U2. We can then right-click and add the two smaller sections, configured for our application. This will launch the Processor Memory Definition dialog that appears in Figure 23.

The dialog box is titled "Processor Memory Definition" and contains a section titled "Physical Memory Block". It has four main input areas:

- Name:** A text box containing "irom". Below it, a description states: "The unique identifier of this memory device. When the FPGA project is compiled these memory details will be passed to the embedded software project. This identifier will also be used to uniquely identify the output HEX file. Names cannot contain spaces."
- Address Base:** A text box containing "0x0". Below it, a description states: "This is the processor's view of where the memory appears in the address space. The size can be specified as a decimal or hex value. Examples: 10000, 0x10000, 1k, 64k, 1M"
- Type:** Two dropdown menus. The first is labeled "Memory Type:" and is set to "ROM". The second is labeled "Memory Type:" and is set to "0 - Fastest". Below them, a description states: "Choose the type and relative speed of the memory device. The linker will use the relative speed settings of the different memories to try and optimize overall performance."
- Size:** A text box containing "16k". Below it, a description states: "This represents the amount of memory that is available to the processor from this device. The size can be specified as a decimal or hex value. Examples: 10000, 0x10000, 1k, 64k, 1M"

At the bottom right, there are "OK" and "Cancel" buttons.

Figure 23. Further configuring the processor memory

From the Configure Processor Memory dialog we will further refine our processor memory by dividing the memory into two 16 Kbyte sections. The table below describes the various options available from within this dialog.

Option	Description
Name	Used to specify a unique identifier for the device. The identifier used for each memory device will be used when generating header files for inclusion in the Embedded Software project. The identifiers will also be used to uniquely identify the corresponding output HEX files. This field <u>cannot</u> contain spaces.
Type	Used to specify the type and relative speed of the memory device. The memory type can be either ROM or RAM (volatile or non-volatile). Speed ranges from 0 (fastest) to 5 (slowest). The Linker uses the speed settings for all defined memories to best optimize the overall performance and efficiency of the code.

Option	Description
Address Base	Used to specify the base address of the memory device. Internal memory will always begin at base address 0000_0000h. The processor's internal memory range is 16MB and the actual size of the physical RAM mapped into this range will be driven by how much physical BRAM is available in the target FPGA. Base addresses can be specified as decimal or Hex notation (e.g. 10000, 0x10000, 1k, 64k, 1M).
Size	Used to specify the size of the memory. This value can be specified as decimal or Hex notation (e.g. 10000, 0x10000, 1k, 64k, 1M).

7.2 Exercise 11 – Configuring the processor memory

In this exercise we will configure the internal memory of the TSK3000 to suit our design.

1. Right-click U2 and select **Configure Processor Memory...** to launch the **Configure Processor Memory** dialog
2. Enable the option to generate a hardware.h (C Header File).
3. Right click U2 in the list of memories and select **Delete Memory**. Once deleted we will need to add our internal memories, this time partitioned as both ROM and RAM for our application.
4. To add the ROM and RAM blocks required, right click the list of memories and select **Add Memory...** This will launch the **Processor Memory Definition** dialog.
5. Specify a new internal memory with the name `iram`, as a **ROM** type, with a Speed of 0 - Fastest, beginning at base address 0x0 and 16 Kbytes in size.
6. Create a second new internal memory with the name `iram`, as a **RAM - Volatile** type, with a Speed of 0 - Fastest, beginning at base address 16k, and 16 Kbytes in size.

Once configured properly, the Configure Processor Memory dialog should appear as it does in Figure 24.

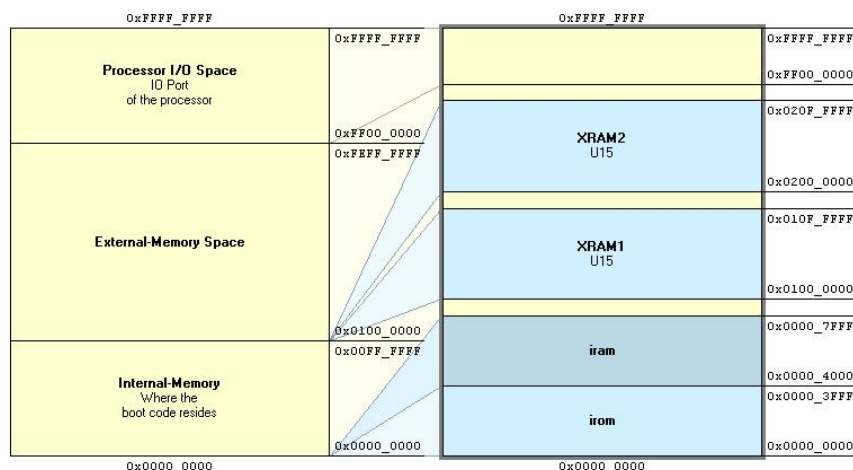


Figure 24. Dividing up the processor memory.

7.3 Configuring processor peripherals

Though we've already specified the details of each peripheral including their bus widths, base address locations, etc. when we had configured the Wishbone Interconnect component U1, we still need to specify this information at the processor end.

To configure processor peripherals, simply right-click the host processor in the schematic and select **Configure Processor Peripheral...** This will launch the **Configure Peripherals** dialog seen in Figure 25.

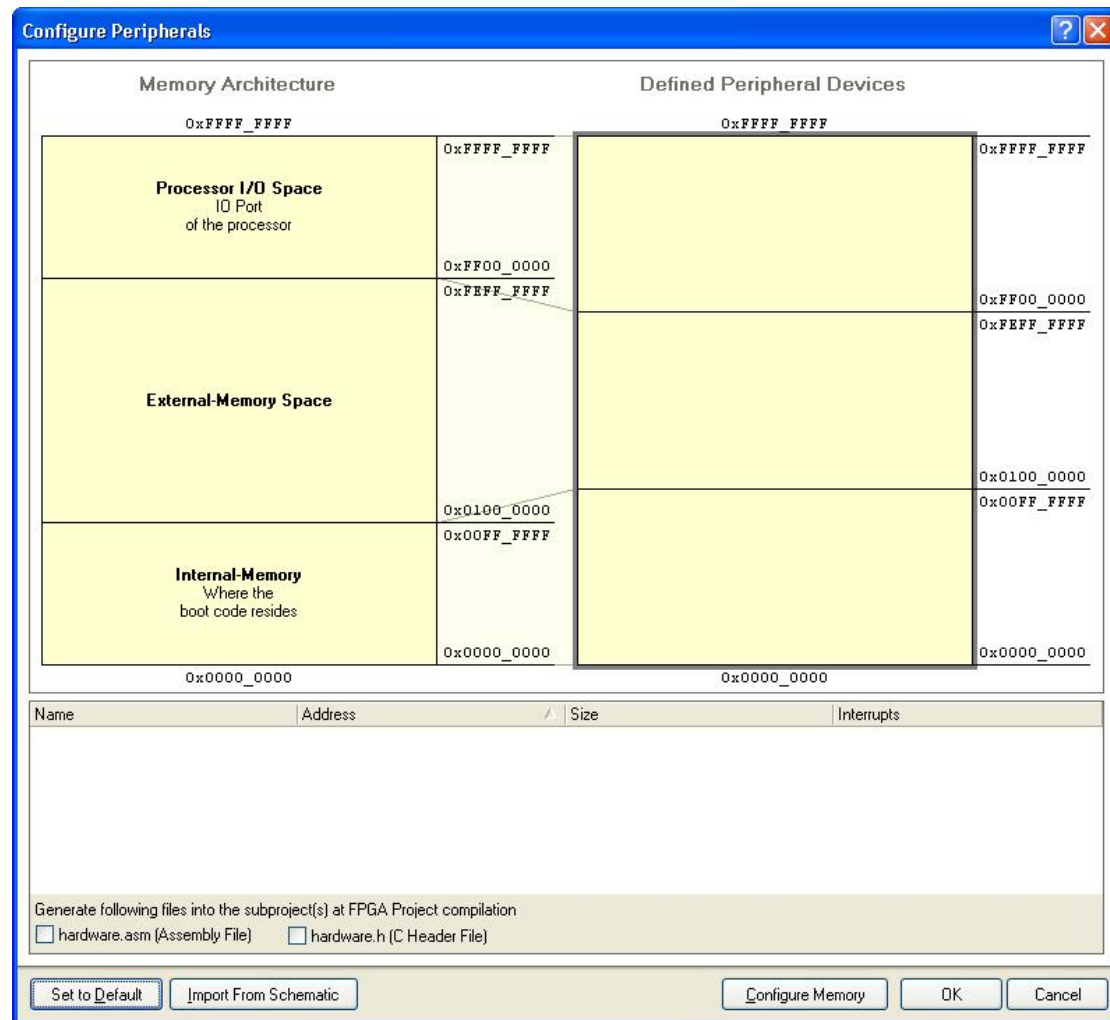


Figure 25. Configuring the processor peripherals at the processor end

Peripherals can be imported into the processor's IO space by selecting the option **Import From Schematic**. This will prompt you with the option to delete existing peripherals before importing, after which, it will launch the **Choose Wishbone Items** dialog seen in Figure 26. Select **Import** under the **Import to Bus** option for any items you wish to import.

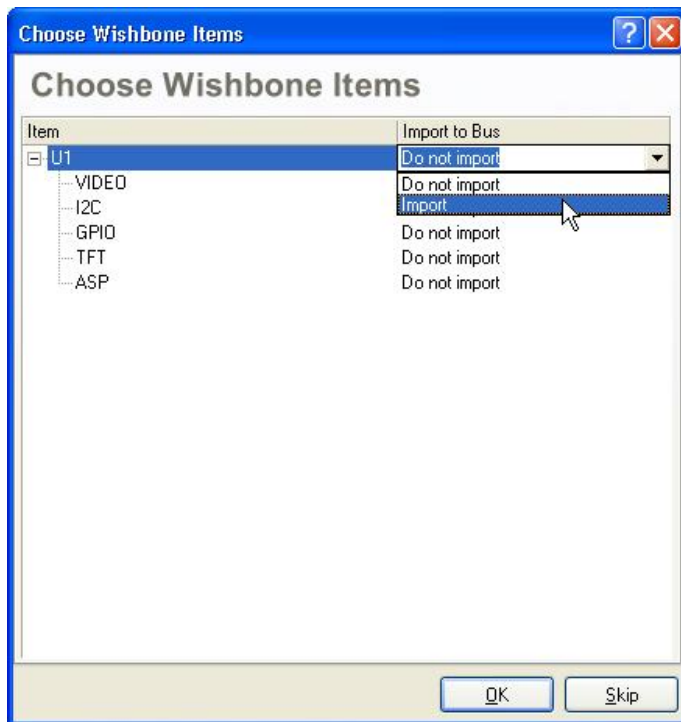


Figure 26. Choose the wishbone peripherals to import

7.4 Exercise 12 – Specifying processor peripherals

In this exercise we will configure the peripheral memory of the TSK3000 to suit our design.

1. Right-click U2 and select `Configure Processor Peripheral...` to launch the `Configure Peripherals` dialog.
2. Enable the option to generate a hardware.h (C Header File).
3. Select the option **Import From Schematic** to import the peripherals from the design. As we've not existing peripherals in our design it is safe to delete these before importing.
4. Select the option **Import** just beside U2 in the `Choose Wishbone Items` dialog. This will cause the status of all items to change to **Import**. Clicking OK will close the dialog and the memory listed in the `Configure Peripherals` dialog should now appear as it does in Figure 27.

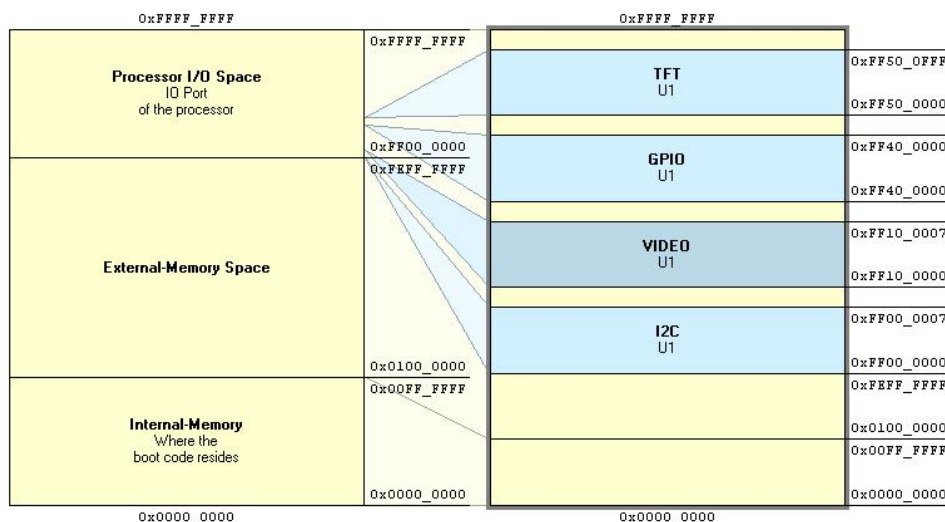


Figure 27. Processor peripherals properly configured for our application.

8 Finalizing the Design, Compiling and Resolving Errors

At this stage we have successfully completed the FPGA portion of our design. It is now time to compile the FPGA project and correct any errors you might have in your design. Once completed, save the FPGA schematic and FPGA project before continuing.

9 Creating Embedded Systems on FPGAs

Altium Designer supports a hierarchical approach to design, even at the project level with support for the linking of multiple projects to one another. This includes the ability to link one or more FPGA projects to a PCB project, as well as the ability to link an embedded project to any FPGA project that uses a softcore processor.

In this section of the course we will explore the creation of embedded projects, the debugging of software code using a software simulator, and some of the many options the TASKING toolset, including the best-in-class Viper compiler technology afford us when designing our applications.

Likewise we will discuss linking embedded projects to an FPGA project and the process of building and running applications running on an FPGA.

9.1 Exercise 13 – Editor Basics

In this exercise we will create a simple embedded project that we will use to explore some of editor options available in Altium Designer. We will also use this project later as we learn the basics of Compiling and Debugging our code.

1. From the **File** menu, select **New»Project»Embedded Project**.
2. Select the newly created embedded project **Embedded_Project1.PrjEmb** in the Projects panel and add a new C source document using **File»New»C Source Document**, or right-clicking the project document in the **Projects** panel and selecting **Add New to Project»C File**.

This will launch a blank text editor that we will use to enter our application code. The buttons along the bottom of the system tray and the menus along the top will change to include options appropriate for developing our application.

3. In the Code editor, enter the following:

```
#define Base_WB_PRTIO_1      0xFF400000
#define Size_WB_PRTIO_1     0x00000001

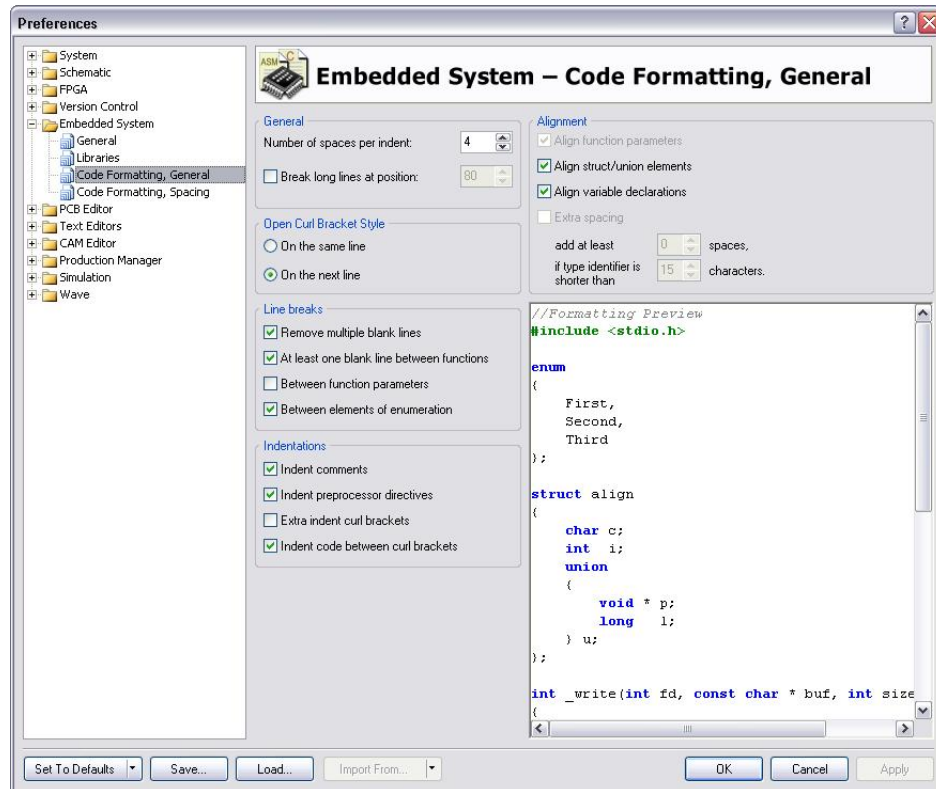
#include "hardware.h"

#define LEDS    (*(unsigned char*)Base_WB_PRTIO_1)

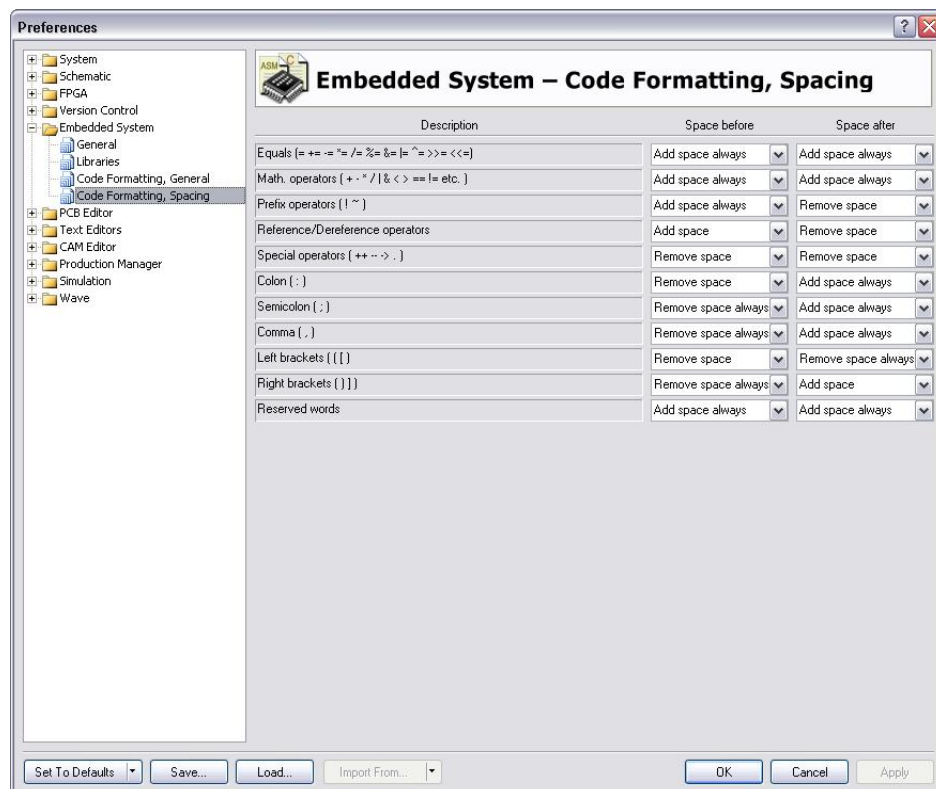
void main(void)
{
    LEDS = 0x55;
}
```

Notice how the code is automatically formatted and the syntax is properly highlighted for the C language. Altium Designer's code editor features a broad range of powerful syntax highlighting capabilities with virtually limitless user configurability available under **Tools»Formatting Options**.

Code Formatting - General

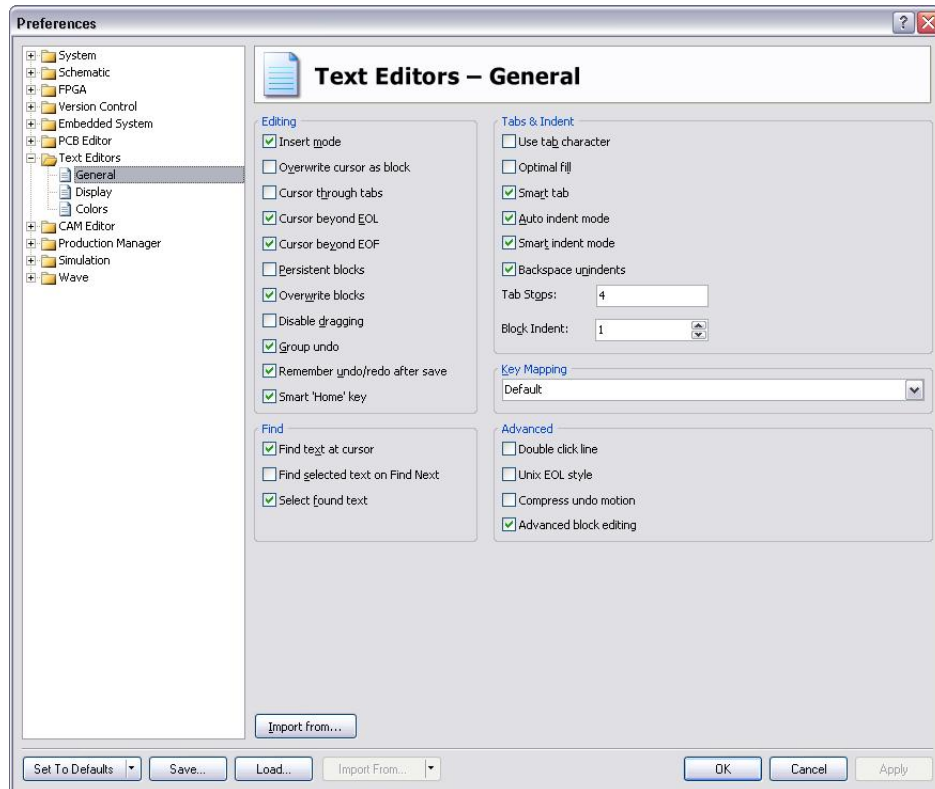


Code formatting - Spacing

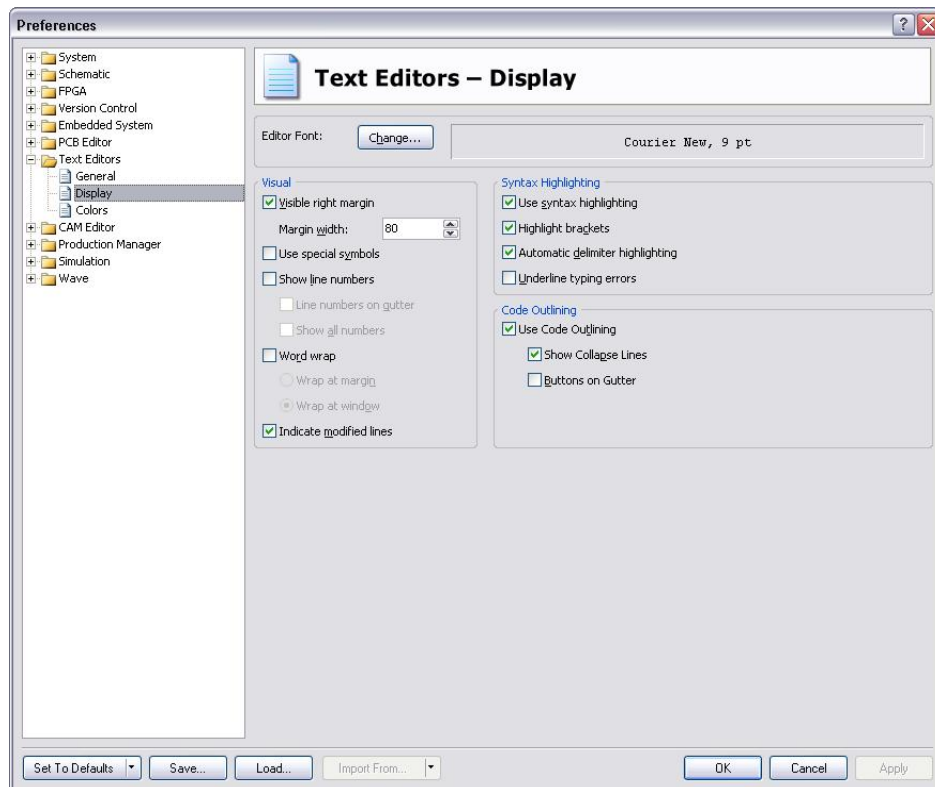


Additional Code Formatting & Highlighting options are available under Tools»Editor Preferences...

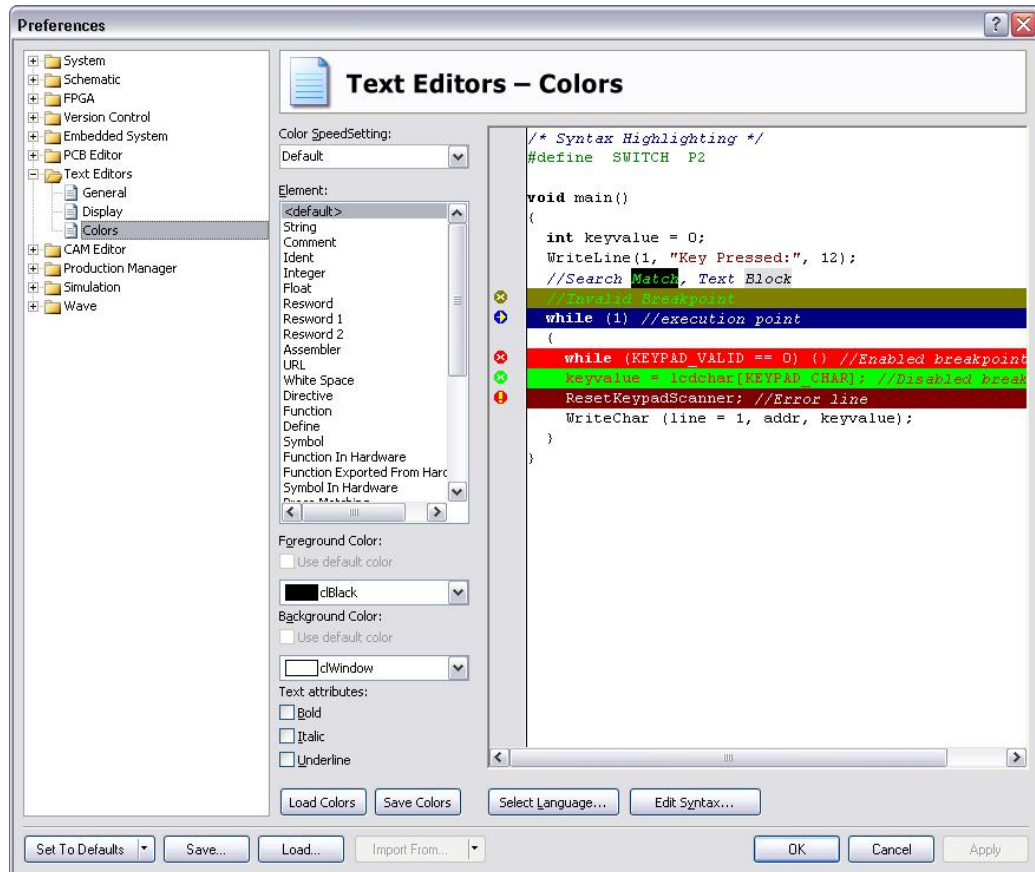
Preferences - General



Preferences - Display



Preferences - Colors



4. If time permits, experiment with some of the editor settings to see their effect on the source code. Use **Tools » Format Source Code** after making each change to update your source code in the code editor.

9.2 The TASKING tool chain

Many long-time users of Altium Designer are familiar with its advanced schematic and PCB capabilities and can understand how schematic-based FPGA development is a natural extension of Altium's core technologies. Delving into the world of embedded systems programming however represents a significant departure from this hardware focus.

Altium, through their corporate acquisition of TASKING, are a major player in the embedded systems marketplace. TASKING products are world-leading tools for embedded software development, bringing together the advanced software design technology needed to compete in the embedded systems marketplace. The award-winning TASKING integrated development environment, compiler, debugger, embedded Internet and RTOS offerings support a wide range of DSPs and 8-, 16- and 32-bit microprocessors and microcontrollers for all areas of embedded development. With over 100,000 licensed users of TASKING products, including the world's leading telecom, datacom, wireless and peripheral manufacturers, the TASKING product range has a long history of technology leadership and innovation.

9.3 The Build flow

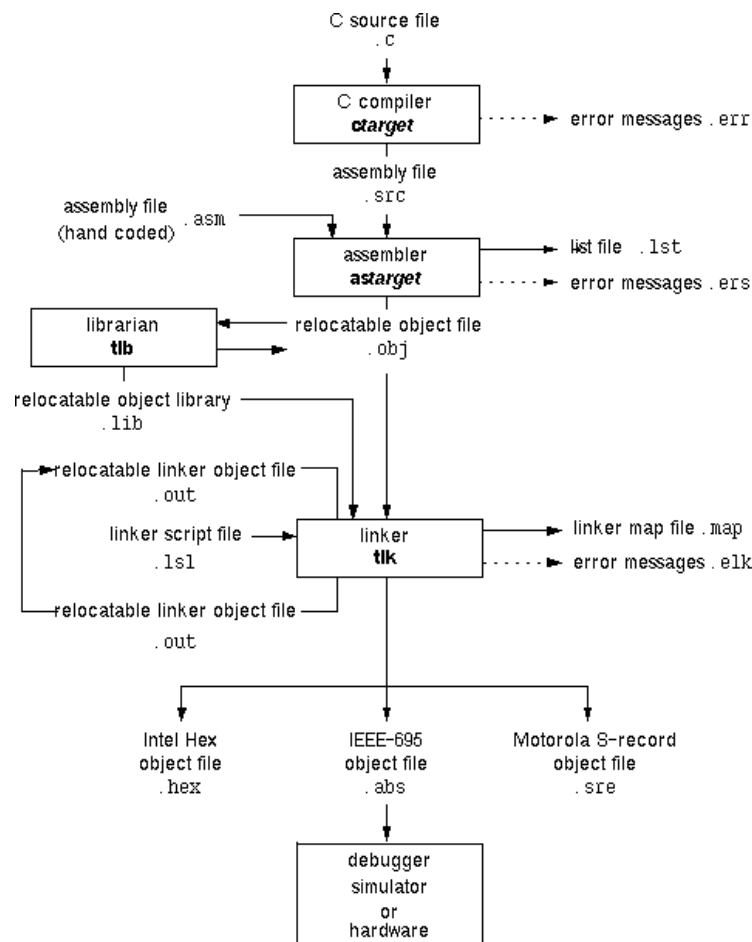


Figure 28. A look at the TASKING build flow.

Building an application is the process of compiling all of the top-level source documents into a binary file that can be executed by a target processor. This is a multi-step process involving a number of tools. In many situations, the user will be shielded from the detail of the underlying compilation processes however in some circumstances it will be necessary to diagnose the source of build or compilation errors and for this it is important to understand the compilation flow.

The C compiler, assembler and debugger are target dependent, whereas the linker and the librarian are target independent. The **bold** names in Figure 28 are the executable names of the tools. Substitute **target** with one of the supported target names, for example, **c3000** is the TSK3000 C compiler and **c51** is the 8051 C compiler.

9.4 Targeting the Project

Prior to compiling an embedded project, it is necessary to notify the compiler which processor we intend to use and any special compilation features we may wish to include. This is done via the **Project Options** dialogue box that can be accessed by right-clicking the Embedded Project in the **Projects** panel and selecting **Project Options** panel or by selecting **Project » Project Options** from the main menu.

9.5 Project options

The **Options for Embedded Project** dialog contains the configuration options specific to the process of compiling, linking, and locating your design. This dialog also contains other options such as the option to specify what processor is being targeted, whether or not to include the Device Software Framework (more on that later in this section), as well as other options.

9.5.1 Device

Select from a range of 8 and 32-bit processors including FPGA-based and non FPGA-based devices. Currently Altium Designer supports the following list of hard and soft processors.

Processor	Description
TSK51	The TSK51x is the core of a fast, single-chip, 8-bit microcontroller, which executes all ASM51 instructions and is instruction set compatible with the 80C31. The TSK51x serves software and hardware interrupts, provides an interface for serial communications and incorporates a timer system.
TSK52	The TSK52x is an 8-bit embedded controller that executes all ASM51 instructions and is instruction set compatible with the 80C31.
TSK80	The TSK80x is a fully functional 8-bit embedded processor which is instruction set compatible with the Zilog Z80CPU. The TSK80x supports hardware interrupts, halt and wait states for low speed memory and I/O devices.
TSK165x	The TSK165x is a fully functional, 8-bit controller that employs RISC architecture with a streamlined set of single word instructions. The TSK165x is instruction set compatible with the PIC16C5X family. All instructions are single cycle, except for program branches which take two cycles.

Processor	Description
TSK3000	The TSK3000A is a 32-bit, Wishbone-compatible, RISC processor. Most instructions are 32-bits wide and execute in a single clock cycle. In addition to fast register access, the TSK3000A features a user-definable amount of zero-wait state block RAM, with true dual-port access. The TSK3000A has been specifically designed to simplify the development of 32-bit systems targeted for FPGA implementation and to allow the migration of existing 8-bit systems to the 32-bit domain with relative ease and low-risk. As a result, complications typically associated with 32-bit system design, such as complex memory management, are minimized.
Actel – COREMP7	Altium Designer's CoreMP7 component is a 32-bit Wishbone-compatible RISC processor, for use in FPGA designs targeting supported Actel Fusion or ProASIC®3 families of physical FPGA devices. Although placed in an Altium Designer-based FPGA project as a CoreMP7, this is essentially a Wishbone-compliant wrapper that allows use of Actel's corresponding 'soft' CoreMP7 processor core. Similar to (and fully compatible with) the ARM7TDMI-S™ core processor, the CoreMP7 is an implementation of the ARM® architecture v4T. This RISC architecture supports both the 32-bit ARM instruction set, as well as the 16-bit Thumb instruction set.
Altera – NIOS II	The Altera® Nios® II is a fully functional, 32-bit load/store, Wishbone-compliant processor that employs RISC architecture with a streamlined set of single word instructions. Nios® II is for use in FPGA designs targeting supported Altera families of physical FPGA devices. The processor comes in three flavors – fast, standard and economy. Although each is placed in an Altium Designer-based FPGA project as a Nios II, this is essentially a Wishbone-compliant wrapper that allows use of Altera's corresponding 'soft' Nios II processor core. All instructions are 32-bits wide and most execute in a single clock cycle (standard and fast variants only). In addition to fast register access, the Nios II features a user-definable amount of zero-wait state block RAM, with true dual-port access. The Nios II core must be licensed from Altera – and must be purchased separately.

Processor	Description
Xilinx - Microblaze	The Xilinx® MicroBlaze™ is a fully functional, 32-bit load/store, Wishbone-compliant processor that employs RISC architecture with a streamlined set of single word instructions. MicroBlaze can be used only with FPGA designs targeting supported Xilinx Spartan® or Virtex® families of physical FPGA devices. Although placed in an Altium Designer-based FPGA project as a MicroBlaze, this is essentially a Wishbone-compliant wrapper that allows use of the 'soft' MicroBlaze processor core. All instructions are 32-bits wide and most execute in a single clock cycle. In addition to fast register access, the MicroBlaze features a user-definable amount of zero-wait state block RAM, with true dual-port access. The MicroBlaze core is licensed as part of the Xilinx EDK (Embedded Development Kit) – and must be purchased separately.
Xilinx – PPC405A	The PPC405A is a fully functional, 32-bit load/store, Wishbone-compliant processor that employs RISC architecture with a streamlined set of single word instructions. As the PPC405 is immersed in a Virtex-II Pro device, only designs targeting this device may make use of the processor. Should you wish the freedom of a device and FPGA Vendor-independent 32-bit system hardware platform, use the available TSK3000A 32-bit RISC processor. Although placed in an Altium Designer-based FPGA project as a PPC405A, this is essentially a Wishbone-compliant wrapper that allows use of the 'hard' PowerPC® (PPC405) processor core immersed in the target physical device. Most instructions are 32-bits wide and execute in a single clock cycle. In addition to fast register access, the PPC405A features a user-definable amount of zero-wait state block RAM, with true dual-port access.

9.5.2 Processor

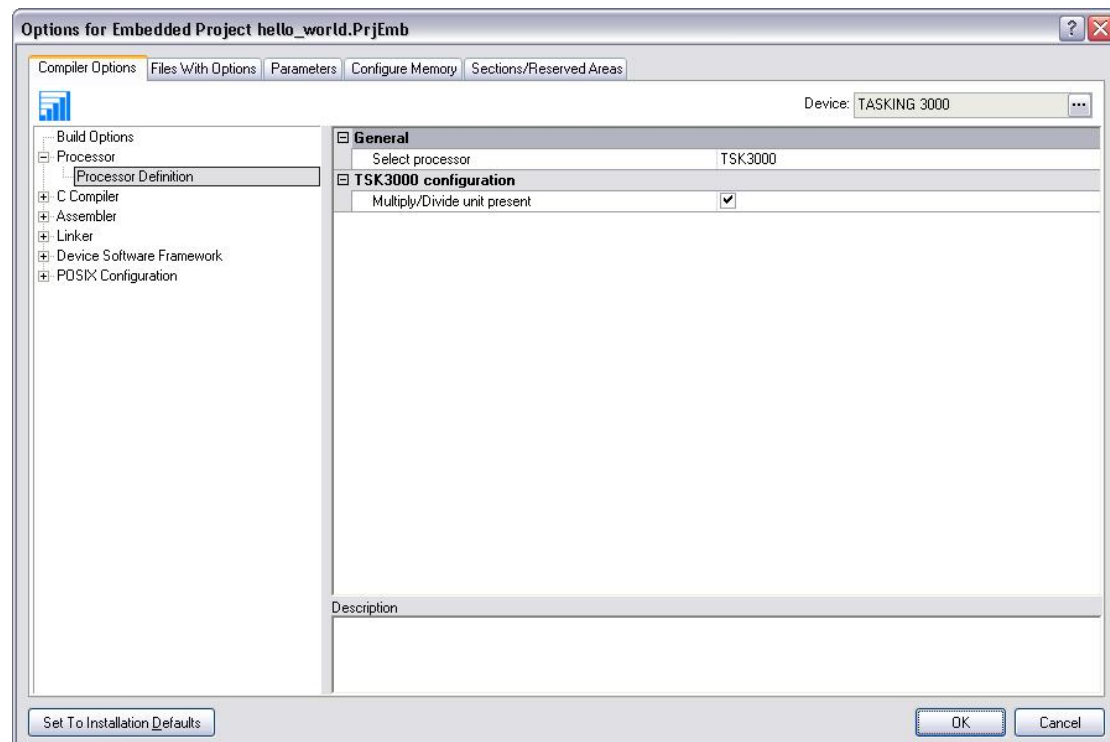


Figure 29. The Processor options in the Project Options dialog

The **Processor** build option has settings for Processor Definition used to specify a particular derivative of a given processor family (for example, Nios IIe, Nios IIc, Nios IIs) if available; as well as processor-specific options to include such things as a multiply / divide unit, floating point unit, etc. depending on the capability of a given processor.

Some processor families also include optional control of the **Startup Code**. The Startup code is used to initialize the processor and any pre-initialized variables *before* execution begins at `void main(void)`. Though the defaults are generally acceptable for the majority of projects, should you ever choose not to include the default startup code produced by the compiler you can disable this option. You will then have to assume the responsibility of handling the reset interrupt, initializing memory variables, and passing control to the `main` function in the C source code.

9.5.3 C Compiler

The Compiler is functionally responsible for parsing the high-level C source commands and reducing them to atomic operations that can be easily mapped onto the target processor's instruction set. This process involves a number of phases including (but not limited to):

1. Preprocessing
2. High-level Optimization
3. Instruction selection
4. Peephole Optimization / Instruction Scheduling
5. Register Allocation
6. Low-level Optimization

Due to the vast variation in requirements and optimization goals, it is not possible to have a “one setting suits all” compiler. Subsequently a number of compiler options are made visible to the user so that the compilation process can be tuned to the specific application. Below is a short list of only a few of these options. For more detailed descriptions of these and more build-related commands, refer to the Embedded Tools Reference for the selected processor family located in the software Help directory or on the Altium website.

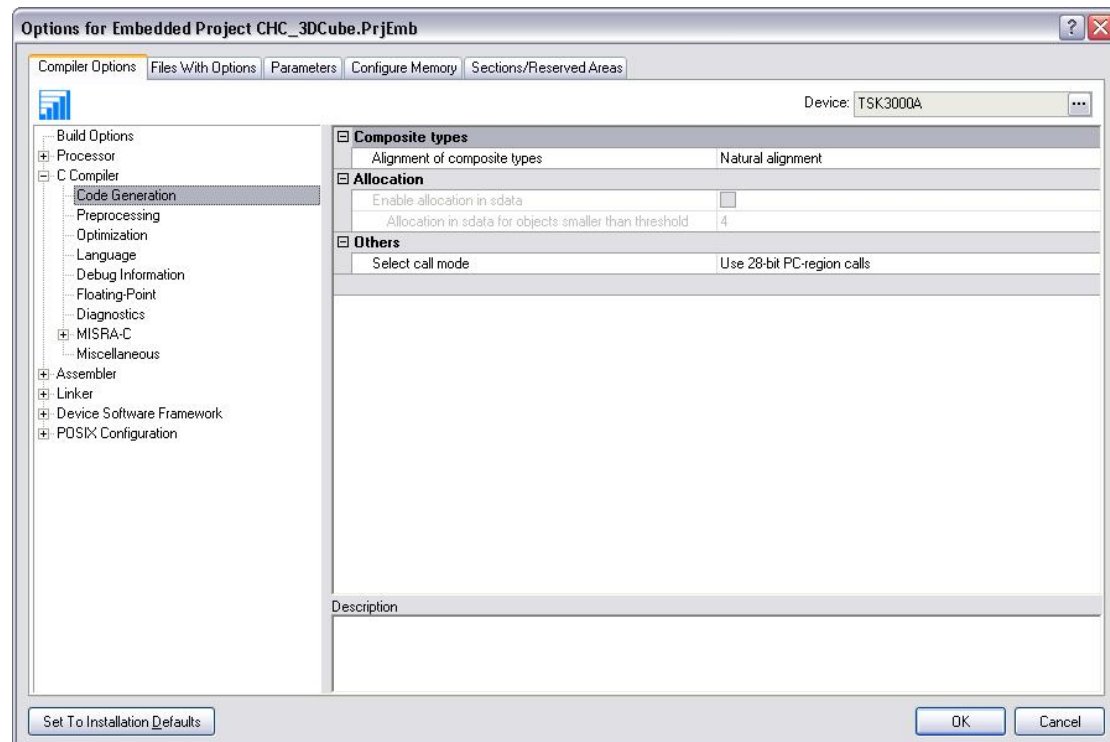


Figure 30. The Compiler options in the Project Options dialog

9.5.3.1 Preprocessing

This section is helpful if it is necessary to define preprocessor macros that will direct conditional compilation in the source. For example you may wish to declare a macro called `DEBUG_MODE` and use conditional compilation in your source code depending on whether this macro was defined:

```
#ifdef DEBUG_MODE
do something...
#else
do normal processing...
#endif
```

This section can also be helpful when changing something across the entire project.

9.5.3.2 Optimization

The TASKING C compiler offers five preconfigured optimization levels as well as a custom level. At each level a specific set of optimizations is enabled. Generally the trade-off is between speed and size. Code can be optimized for either speed or size however that code optimized for speed is generally larger, whereas code optimized for size is generally slower. Applications that have not been optimized will generally execute more quickly; however the amount of space required in memory will be more substantial. There is also the trade-off with respect to the ability to debug. Highly optimized code is difficult to debug as the relationship between the source code and the compiled code becomes progressively more difficult to understand.

It is suggested to begin with the fewest possible optimizations enabled and after ensuring the code is correct, progressively optimize your application until achieving the ideal relationship between speed and size.

- **Level 0:** No optimizations are performed. The compiler tries to achieve a 1:1 resemblance between source code and compiled code. Expressions are evaluated in the order written in the source code, associative and commutative properties are not used. Compiled code will generally run faster at this level of optimization and performance will decrease as the software is further optimized.
- **Level 1:** Enables optimizations that do not affect the debug-ability of the source code. Use this level when you are developing/debugging new source code.
- **Level 2:** Enables more aggressive optimizations to reduce the memory footprint and/or execution time. The debugger can handle this code but the relationship between source code and generated instructions may be hard to understand. Use this level for those modules that are already debugged. This is the default optimization level.
- **Level 3:** Enables aggressive global optimization techniques. The relationship between source code and generated instructions can be very hard to understand. Use this level when your program does not fit in the memory provided by your system.
- **Level 4:** Fully optimize for size. This is the most aggressive level of optimization and should only be used once the application has been completely debugged. The application may execute more slowly however its memory footprint can be far less substantial.
- **Custom level:** you can enable/disable specific optimizations.

9.5.3.3 Language

Default settings are usually adequate however occasionally you may wish to build code that is pre ISO C 99 compatible and will therefore need the options contained in this panel. This panel also contains options to allow use of GNU C extensions if desired.

9.5.3.4 Debug information

In general it is helpful to always generate debug information unless building for a final production release where debug information would be superfluous. You might also disable this option if you intend to make your code available as a library and would prefer to protect your IP.

9.5.3.5 Floating point

In general the default settings are acceptable though the option to **Use single precision floating-point only** can affect both the size and execution speed. When checked, variables of the type 'double' will be treated as 'float'. Select this option in the event you do not require double-precision.

9.5.3.6 Diagnostics

This section controls how compilation warnings are reported. In some cases it may be desirable to suppress specific warnings if they are creating too much 'noise' in the **Messages** Panel.

9.5.3.7 MISRA C & MISRA C Rules

The Motor Industry Software Reliability Association (MISRA) is in existence "To provide assistance to the automotive industry in the application and creation within vehicle systems of safe and reliable software." Through extensive consultation within the automotive industry, MISRA has completed the development of guidelines specifically aimed at the use of the C language in safety related systems. These guidelines primarily identify those aspects of the C language that should be avoided in safety-related systems, along with other

recommendations on how other features of the language should be used. It is anticipated that the guidelines will be adopted for embedded C programming throughout the automotive industry.

Altium Designer includes a number of compilation options that can flag as a warning code that does not comply with MISRA recommendations.

9.5.3.8 Miscellaneous

Use this section to pass any compiler flags or settings that have not been covered in the previous panels. The **Options String** at the base of the compiler settings panel provides an indication of the options that will be passed to the C compiler. Further information about each individual setting can be found in **GU0105 Embedded Tools Users Guide.pdf** or via the help system under **Embedded Software Development » Embedded Tools Options Reference » Compiler Options**.

9.5.4 Assembler

The assembler converts hand-written or compiler-generated assembly language programs into machine language, using the IEEE-695 object format. These files serve as input for the linker.

9.5.4.1 Phases of the assembly process

1. Preprocess directives
2. Check syntax of instructions
3. Instruction grouping and reordering
4. Optimization (instruction size and jumps to branches)
5. Generation of the relocatable object file and optionally a list file

The **Project Options...** dialogue box contains a number of assembler options. The subsections of the assembler options allow for additional control over the assembler in much the same way that the previously mentioned compiler options do.

The default options are generally sufficient for most applications. However, should you find it necessary to tune the assembler; further information can be found in the Embedded Tools Reference for the target processor available under the help system or on the Altium website.

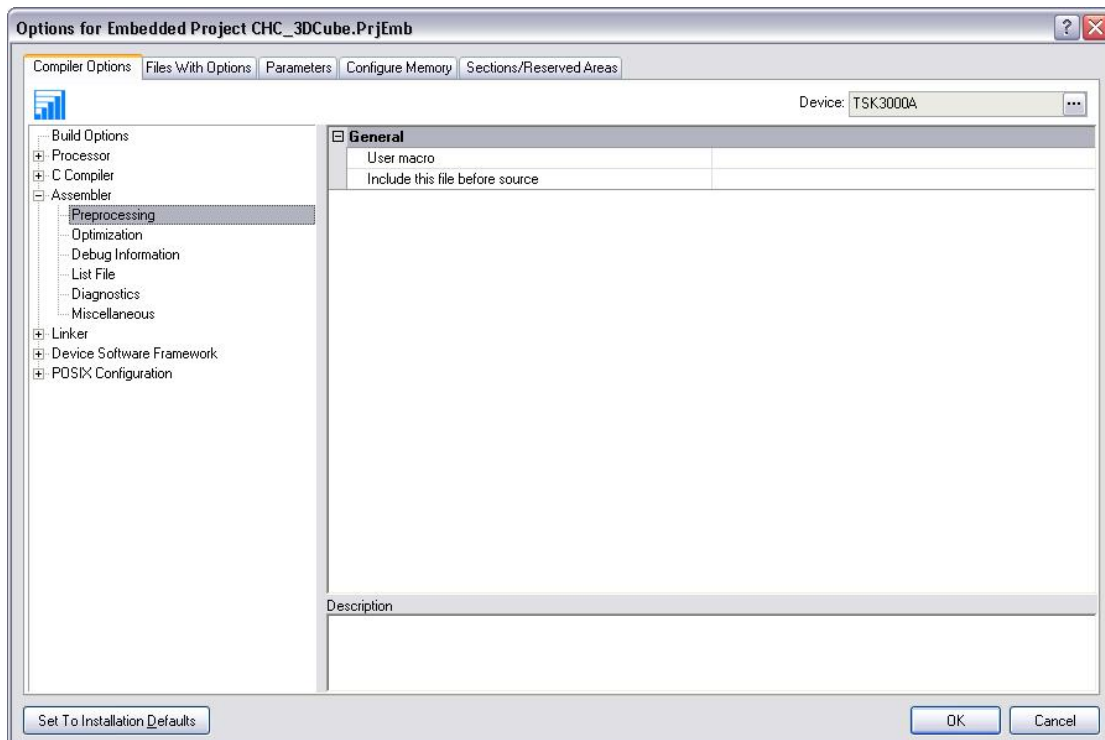


Figure 31. The Assembler options in the Project Options dialog

9.5.5 Linker

The linker combines and transforms relocatable object files (.obj) into a single absolute object file. This process consists of two phases: the linking phase and the locating phase.

In the first phase the linker combines the supplied relocatable object files (.obj files, generated by the assembler) and libraries into a single relocatable object file. In the second phase, the linker assigns absolute addresses to the object file so it can actually be loaded into a target.

The linker can simultaneously link and locate all programs for all cores available on a target board. The target board may be of arbitrary complexity. A simple target board may contain one standard processor with some external memory that executes one task. A complex target board may contain multiple standard processors and DSPs combined with configurable IP-cores loaded in an FPGA. Each core may execute a different program, and external memory may be shared by multiple cores.

Most linker options can be controlled via the project options dialog but some options are only available as command line switches. The default options are generally sufficient for most applications however should you find it necessary to tune the linker then further information can be found in the Embedded Tools Reference for the target processor available under the help system or on the Altium website.

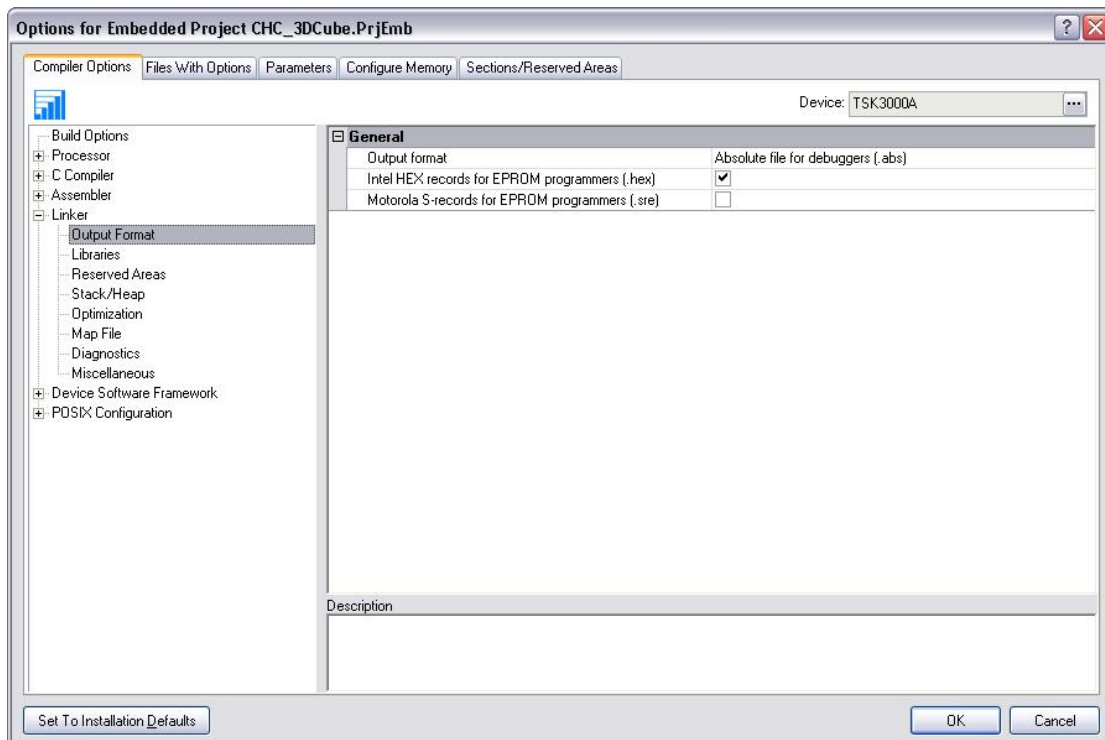


Figure 32. The Linker options in the Project Options dialog

9.5.6 Device Software Framework

Altium Designer's Device Software Framework (DSF) has been designed to dramatically simplify embedded application development in the Altium Designer environment.

The Device Software Framework delivers:

- A Low Level Peripheral Interface (LLPI) layer, with tight integration between the FPGA peripherals and their driver code.
- A Processor Abstraction Layer (PAL) greatly simplifies the portability of your embedded application across all target processors, both embedded and discrete 32-bit, supported by Altium Designer.
- While the DSF 'abstracts away' most of the physical layer from the embedded application development process, you continue to have direct access to the processor's interrupt system.

Included in the Device Software Framework are the low-level drivers required to support the peripheral cores bundled with Altium Designer. The availability of the Processor Abstraction Layer ensures that when utilizing these peripherals, it is possible to change 32-bit processors with limited impact to the overall system.

Device Software Framework source files (the low level drivers supplied with Altium Designer) are stored in the application directory under `C:\Program Files\Altium Designer\6\System\Tasking\dsf\`

Here you will find all of the C source files as well as their associated header files. Though these files will not be copied to the project upon compile, they will be included if the option to utilize the Device Software Framework is enabled.

To use the Device Software Framework and the associated driver files, you must include the following lines in a C source file within your embedded project:

```
#define DSF_IMPLEMENT
#include "dsf_system.h"
```

This will enable the Low Level Peripheral Interface. To enable the Processor Abstraction Layer you must check the embedded project option **Use Processor Abstraction Layer in project**. This will make the design more easily portable between processors as this will abstract the interface of the processor from the peripherals you include in the design.

Figure 33 highlights some of the differences between traditional embedded systems and those which utilize DSF.

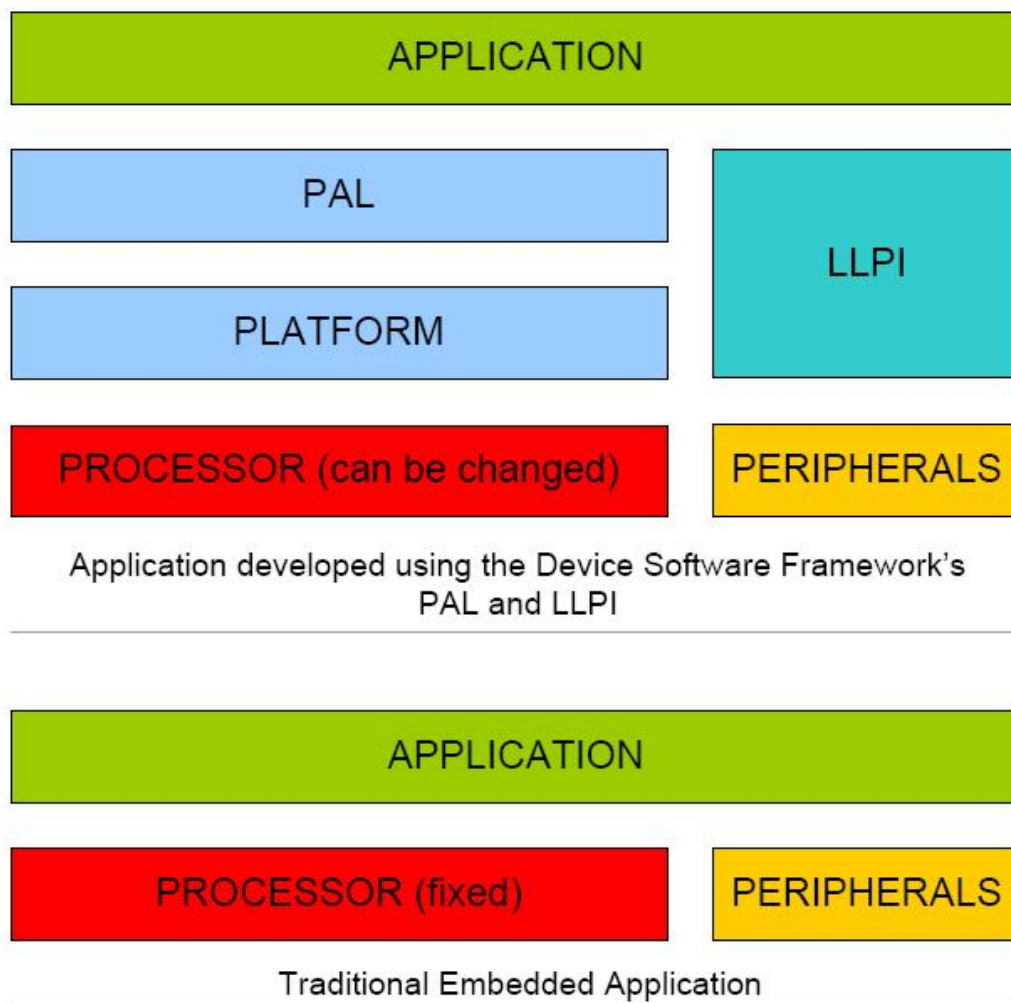


Figure 33. Differences between a traditional embedded application and Device Software Framework

The DSF allows the embedded application to be portable between processors, instead of being tightly coupled to the hardware as it is in the traditional approach.

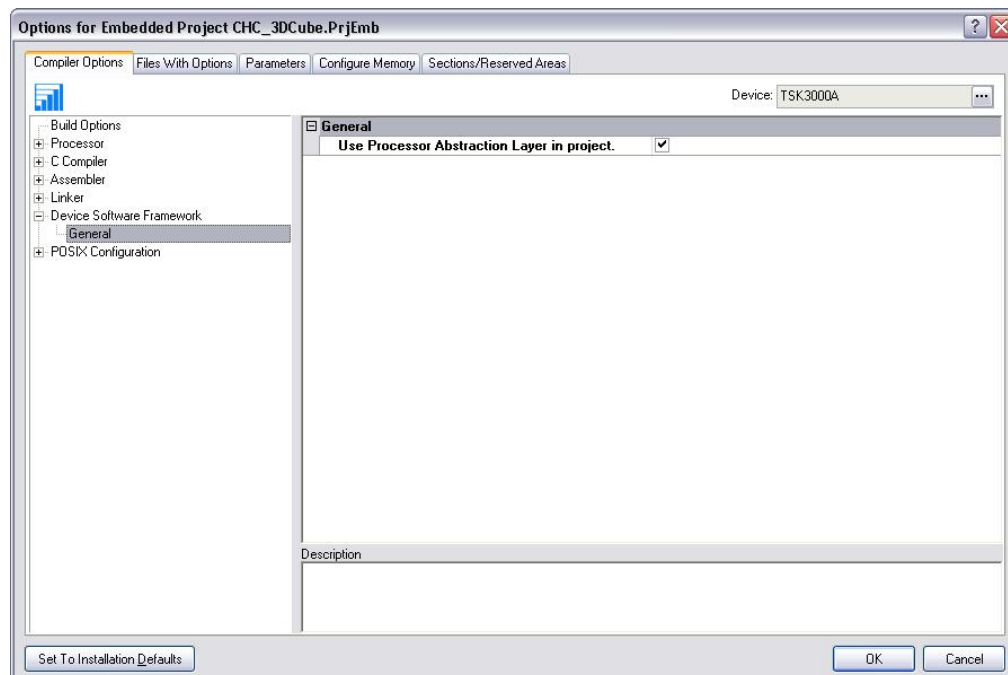


Figure 34. The Device Software Framework options in the Project Options dialog

9.5.7 POSIX Configuration

Altium Designer includes support for a POSIX compliant Minimal-Real Time kernel, supporting the POSIX.13 PSE51 Real-Time profile. This means that the implementation supports the required interfaces referenced in the appropriate standardized profile. These interfaces support the functional behavior and any additional constraints or options as described in the appropriate base standard.

To utilize the POSIX kernel, check the option **Compile project as a POSIX application** and select the **Units of Functionality**, **Option Requirements**, and **Implementation Defined Constants** required by your application.

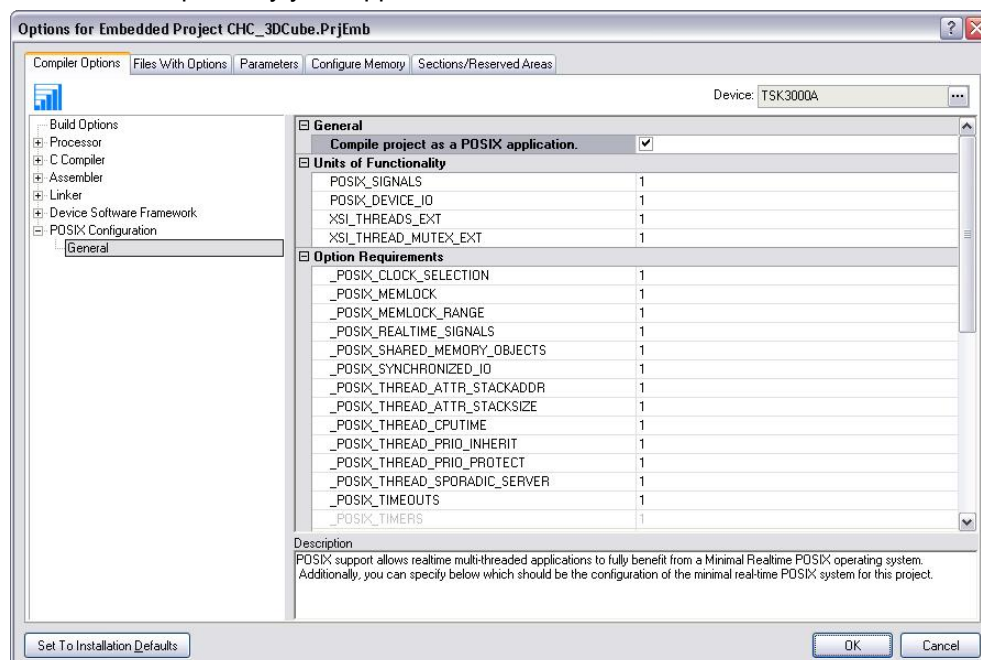


Figure 35. The POSIX options in the Project Options dialog

9.6 Attaching an embedded project to an FPGA project

Creating a link between the softcore and its associated embedded code ensures that correct code / processor coupling is made. It also ensures that subsequent project builds incorporate all hardware and software changes that have been made.

The process of linking an FPGA Project to its associated embedded code first requires that both projects are open together. Figure 36 shows two projects open together prior to being linked. These two projects are not bound in any way.

To link the Embedded Project to the FPGA Project we must utilize the **Structure Editor**.

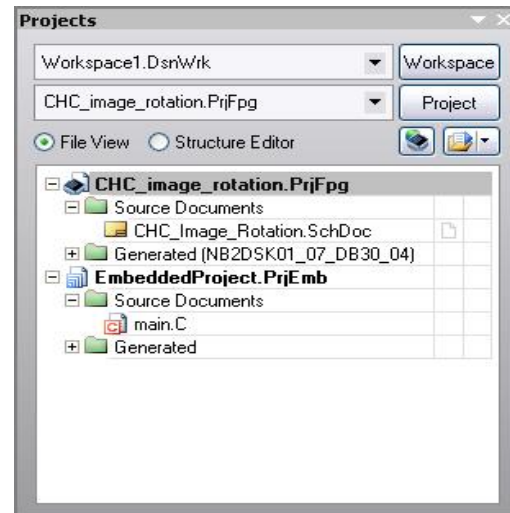


Figure 36. Unlinked projects

File View

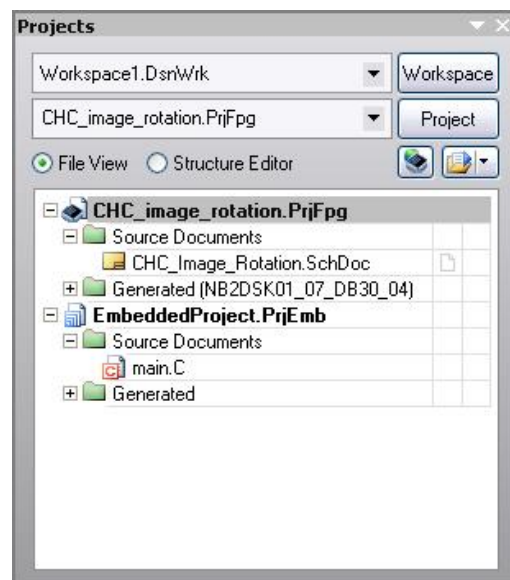


Figure 37. Linked projects (File View).

In the **File View**, files are grouped primarily according to which project they are a part of and secondarily according to the file type – schematic, PCB, settings, etc.

Structure Editor

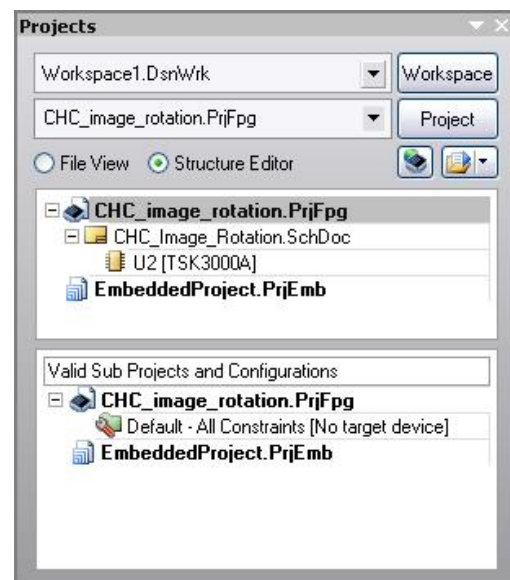


Figure 38. Linked projects (Structure Editor).

In the **Structure Editor**, the hierarchical linkage between projects is shown – i.e. above we see that the embedded project has been linked to U3 (the TSK3000A core) in the FPGA project.

9.6.1 Linking the projects

Linkage between projects is created and broken using drag-and-drop. By click-and-holding the left mouse button on a sub-project, all possible drop locations (valid linkage points) will highlight in blue and the sub-project can be drag-and-dropped onto the parent project to create the link.

To break the linkage, drag the sub-project away from the parent project and drop it on a clear region of the **Structure Editor**.

Alternatively you can right click on the processor and specify the embedded project using the **Set Embedded Project** menu option.

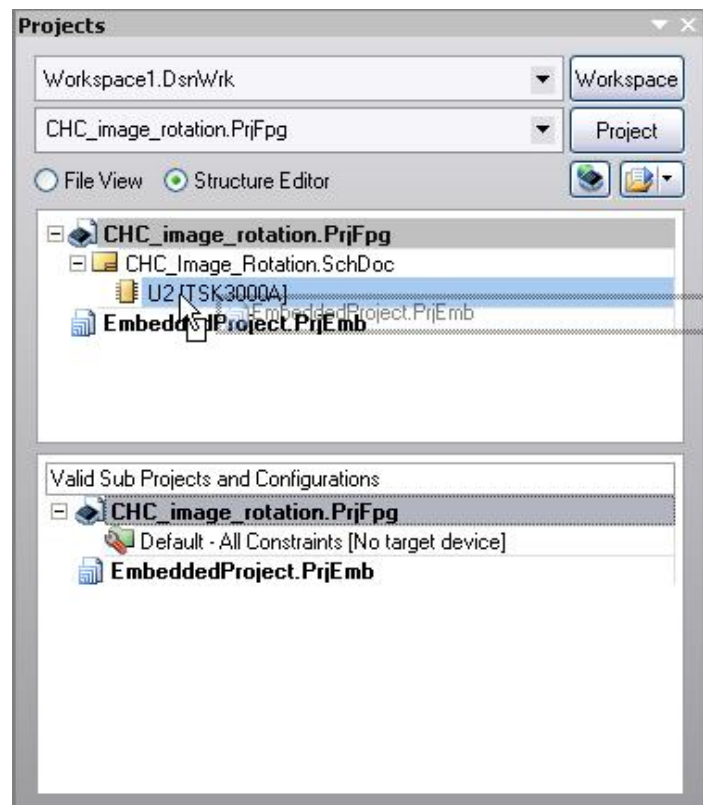


Figure 39. Linking an embedded project to its hardware

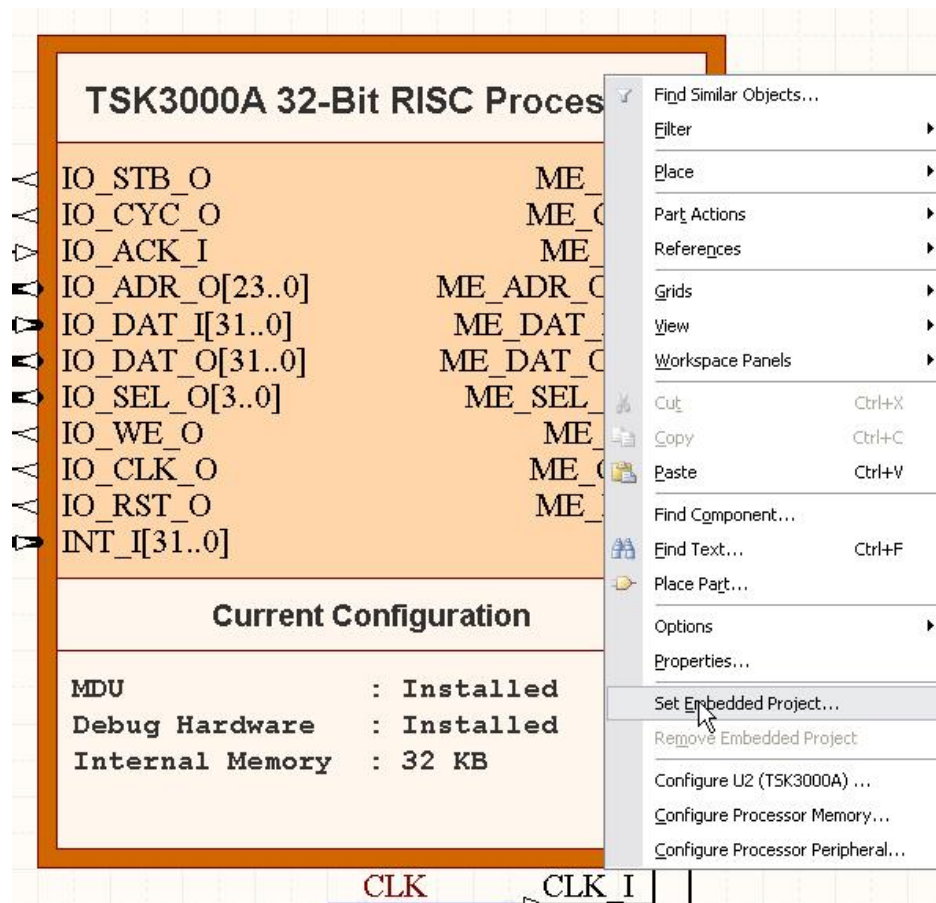


Figure 40. Linking an embedded project to a processor via the schematic interface

9.7 Exercise 13 – Linking projects

In this exercise we will link our two projects to one another.

1. With both your FPGA and Embedded projects loaded in the **Projects** panel change to the **Structure Editor** view.
2. Link the two projects as described in the previous section.
3. Once the projects have been correctly linked return to the File View and the projects should appear linked.
4. You can try *unlinking* the two projects again by dragging the linked embedded project off of U2 in the structure editor.
5. Save your work.

9.8 Exercise 14 – Building our design

In this section we will configure our design for the build process and subsequently build both our FPGA project and embedded projects, programming the Spartan 3 FPGA on the Desktop NanoBoard.

1. Auto Configure the FPGA project.
2. Right Click the FPGA Project in the Projects panel and Select Add New to Project»Constraint File to add a new blank Constraint file to the project.
3. In the constraint file, add a new port constraint that constraints CLK_BRD with a constraint kind of FPGA_CLOCK_FREQUENCY with a value of 50MHz.

Record=Constraint TargetKind=Port TargetId=CLK_BRD FPGA_CLOCK_FREQUENCY=50MHz

4. Make sure your Desktop NanoBoard is connected to your PC and powered on.
5. Ensure that the **Live** checkbox is checked. You should see a picture of the Desktop NanoBoard in the upper region of the display and an icon of the Spartan3 FPGA in the middle region.
6. In the drop down list just below the Spartan3 icon, ensure that the **CHC_image_rotation / NBD2DSK01_07_DB30_04** project / configuration pair is selected.
7. Click once on the words **Program FPGA** to begin the build process.
8. As the build process progresses, the colored indicator from each stage will turn yellow while it is processing and then green when completed successfully. The process of building the design may take several minutes to complete. You can observe the progress of the build from the **Messages** and **Output** panels which can be accessed from the **System** Panel tab in the lower right section of the Altium Designer workspace (see Figure 41).

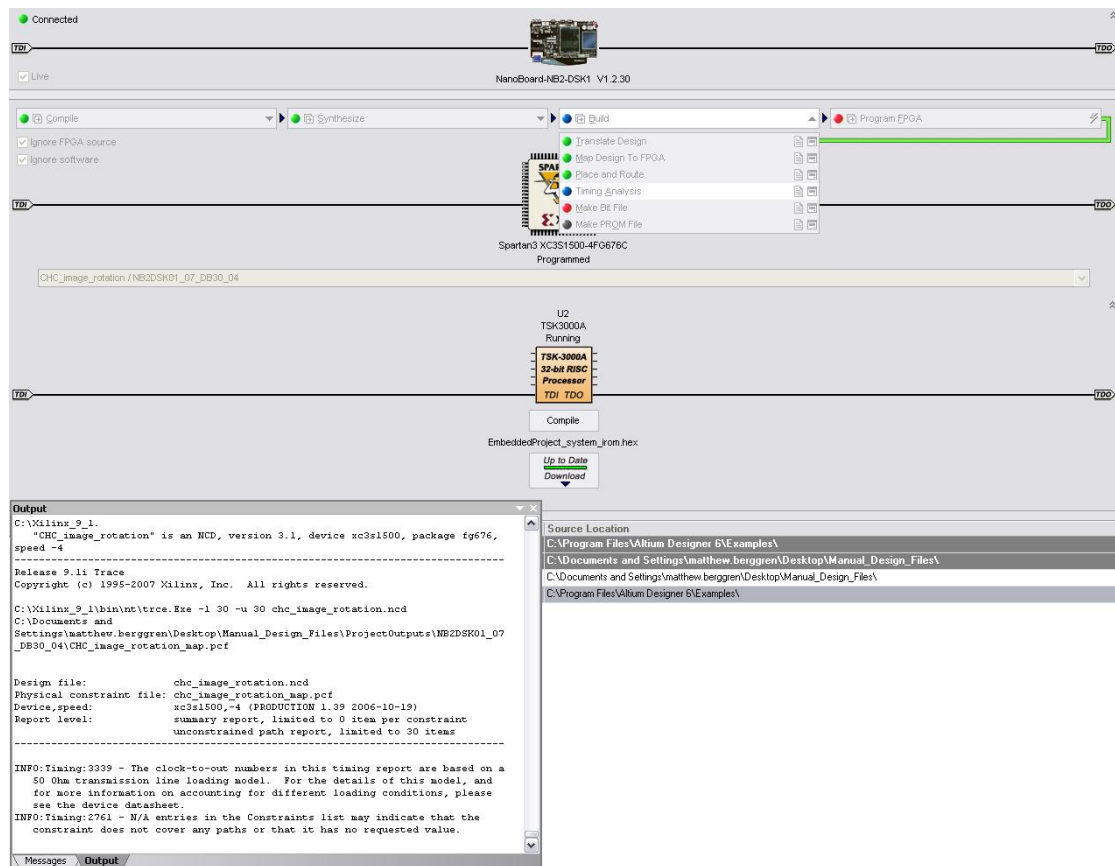
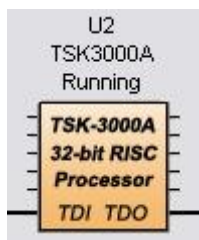


Figure 41. Building the FPGA design

9. If any errors occur you will need to rectify them before you can proceed. Try to locate the source of the error by retracing your steps through the instructions of the tutorial.
10. A summary dialog will be displayed once the design has been built and downloaded successfully. Click **OK** to close this dialog.

Once the FPGA design has been downloaded to the Desktop NanoBoard, you should notice that the status of the TSK3000 processor has changed from **Missing** to **Running**.



Results Summary

?

✕

Device Resources - Usage Summary

4-Input LUTs - Logic	5,230 / 26,624	19%
Block RAMs	17 / 32	53%
Global Clocks	3 / 8	37%
I/O Pins	203 / 487	41%
MULT18X18s	4 / 32	12%
Slice Flip Flops	2,740 / 26,624	10%
Slices with only related logic	3,672 / 3,672	100%
Slices with unrelated logic	0 / 3,672	0%
Slices	3,672 / 13,312	27%
Total 4-Input LUTs - Logic	5,971 / 26,624	22%

Design Statistics - Timing Summary

Maximum Frequency	36.346 MHz
Maximum combinational path delay	8.760 ns
Minimum input required time before clock	7.372 ns
Maximum output delay after clock	33.725 ns
No timing constraints.	

☒ Show Results Summary dialog
 Note: The Results Summary also appears in the Output panel

Print...

Copy

Report

Close

11. At this stage our design has been downloaded to the NanoBoard and the LEDs should display a binary pattern consistent with the hex value 0x55 that we had specified in our source code (left to right the LEDs should read 01010101_b or 85_d with the LSB being the right-most bit in the row of LEDs).



9.9 Exercise 15 – Updating our software Live

In this exercise we will make changes to the software running on the host processor inside the FPGA on the NanoBoard, then update these software changes to the processor as it executes in system, in real-time.

1. With the FPGA and Processor successfully programmed on our NanoBoard, open the main.C file in the EmbeddedProject.PrjEmb file.
2. Modify the function `main` to read:

```
void main(void)
{
    LEDS = 0xFF;
}
```

3. Updating the software running on the NanoBoard does not require us to reprogram the FPGA, only to update the software on running on the processor.

Press the **Compile and Download** button  in the toolbar at the top of the screen to recompile the software and update your changes to the processor running on the FPGA.

This should change the state of the LEDs on the NanoBoard such that they appear to all be on.

10 Updating the Design to Display Video

In this section we will modify the software to read video from the composite video input and output that video on the touch screen display. We will then explore some of the software debug capabilities found in the Altium Designer environment and employ our virtual instruments to test our design Live, in-system.

10.1 Defining Sections

It is sometimes useful to define memory sections in embedded software. Sections are a reference to a specific location in memory that can be passed to the locator to control the way that a program is located or uses memory.

Sections are created from the **Project Options** dialog for the embedded project. To create a section, launch the **Project Options** dialog and select the tab **Sections/Reserved Areas**. The **Add Section** button is used to add a new section and will launch the **Section** dialog. The **Add Reserved Areas** button is used to create new reserved areas that or areas that will not be used during the Linking and Locating process.

The screenshot shows the 'Section' dialog box with the following fields and options:

- Name:** A text box containing '.bss:video'. Below it is explanatory text and examples for C and ASM code.
- Location:** A text box containing 'mem:XRAM1'. Below it is explanatory text and examples for decimal, hex, and memory names.
- Fill Bit Pattern:** An empty text box. Below it is explanatory text and an example '0xAA'.
- Locate Static Data:** An unchecked checkbox with explanatory text.
- Dependencies:** An unchecked checkbox labeled 'Include Dependencies' with explanatory text.
- Notes:** A large text area at the bottom.
- Buttons:** 'OK' and 'Cancel' buttons at the bottom right.

Figure 42. Creating sections for the locator

The following table details the various options in the **Section** dialog.

Option	Description
Name	The name field is used as a label or reference to the section defined in the source code.
Location	This is the processor's view of where the section appears in the address space. The size of the section is automatically allocated. The location can be specified as a decimal or hex value, or a memory.
Locate Static Data	Used to locate the ROM initialization data for the section. Uncheck to locate the section itself.
Fill Bit Pattern	Used to initialize the empty spaces at the end of a section as a result of an MAU alignment. If the section is in ROM, leaving this blank will fill the section with zeros. If it is in RAM, the spaces are not initialized.
Dependencies	If checked, any dependencies of the section will also be located by this section layout directive.
Notes	Used to add any user notes that might help explain or describe the section.

10.2 Exercise 16 – Updating our software Live

In this exercise we will create two sections, the names of which will be labels that can then be used as a qualifier when we create an array for the video capture and video display buffers.

The advantage to Sections is in the finer control it gives over the way in which a program is Located and uses memory. Sections can be created and even nested to make an almost 'hierarchical' partitioning of a design's memory resources possible.

1. Right-click the embedded project in the **Projects** panel and select **Project >> Project Options**.
2. Select the **Configure Memory** tab and observe that entries for XRAM1 (the Video Capture side of our design) and XRAM2 (the Video Display side of our design) exist.
3. Select the Sections/Reserved Areas tab of the **Options for Embedded Project CHC_Image_Rotation.PrjEmb** dialog.
4. Click the **Add Section** button to launch the Section dialog.
5. In the Section dialog box:
 - a. Set the **Name** to **.bss.video**
 - b. Set the Location to **mem:XRAM1**
6. Click OK to close the Section dialog box and observe that the new memory section has been added.
7. Click the Add Section button again
8. In the Section dialog box:
 - a. Set the Name to **.bss.tft**
 - b. Set the Location to **mem:XRAM2**
9. Click OK to close the Options for Embedded Project CHC_Image_Rotation.PrjEmb dialog.

10.3 Exercise 17 – Updating our Software to Capture Video

In this exercise we will update our application code with the software required to capture and display video. Utilizing the Device Software Framework, much of the process of initialization and the low-level interaction with the hardware will be simplified. We instead will communicate with the hardware through its Low Level Peripheral Interface layer we'd discussed earlier. This dramatically reduces the amount of effort required in getting our system off the ground.

1. Open main.C and remove any existing lines of code that might be in the file.
2. Launch the Snippets panel under System»Snippets using the buttons along the lower right corner of the workspace.
3. Locate and Add the Snippet "Video_Pass_Through_1" to main.C. This contains the #defines that specify the basic characteristics of the video including its height, width, screen resolution (both X and Y) and the attributes required to properly center the image on the screen.

This section also contains #pragmas for both the video and display memories, used to define the size of the Sections required by these two buffers.

4. Return to the Snippets panel and add the Snippet "Video_Pass_Through_2" to main.C (the position of the cursor in the text editor will determine the point at which the Snippet is inserted).
5. If the code appears unformatted, select Tools»Format Source Code and it will be formatted automatically to match your formatting standards specified under Tools»Formatting Options...

Examine the source code in the function `void main(void)`. The code is quite simple really.

6. Holding the Ctrl key on your keyboard, hover your cursor over the various function names listed in the initialization sections. You will see the cursor change to a small hand.
7. Clicking on the name of one of these functions whilst in this mode will open the source Device Software Framework source file containing the function being referenced. We are essentially calling a series of DSF functions and passing them parameters that simplify the initialization and control of the behavior of the peripherals.

```
//Main Processor clock rate
_ticks_per_sec = 50000000; // Assuming we run at

//Initialize TFT display
vga16_init(& display_settings, Base_TFT, (uintptr_t)0);

/* I2C */
i2c_init(Base_I2C);

// Initialize video capture
bt656_init(& capture_settings, Base_VIDEO, Base_I2C);
bt656_set_linesize(& capture_settings, VIDEOWIDTH);
bt656_set_scale(& capture_settings, 1, 1);
bt656_set_framerate(& capture_settings, 1);
bt656_set_crop(& capture_settings, (PALWIDTH - VIDWIDTH)/2, (VIDHEIGHT - VIDEOHEIGHT)/2, VIDWIDTH, VIDEOHEIGHT);
bt656_set_buffer(& capture_settings, (uint32_t *)0);
bt656_set_mode(& capture_settings, BT656_RUN);
```

8. Rebuild and download your source code by hitting the  button in the toolbar along the top of the workspace. This will automatically save the source file at the same time.

Video should now display from your camera on the screen on your NanoBoard. Try moving the camera and have a look around.

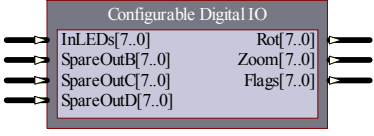
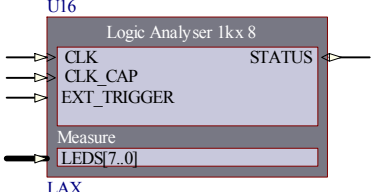
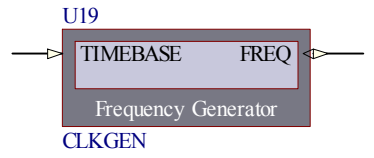
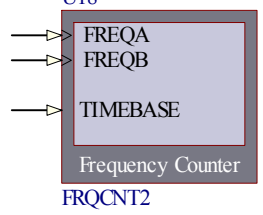
10.4 Utilizing the Virtual Instruments

To achieve an understanding of how our design might function in its final implementation – i.e. the “real-world,” Altium Designer provides a host of Virtual Instruments that can, utilizing FPGA-resources, be used to test and debug our system live. A major advantage of this approach is that the interaction is with the real hardware and not a software simulator. This allows us to find and correct logic errors, as well as issues with the software, on the physical hardware, in real-time.

Further to this, utilizing this methodology and the inherently reprogrammable nature of the FPGA platform, it is easy to see how we might later use instrumentation in our design, even *after* it has been fully implemented, to test our design, in-system, and possibly even other components of our system.

10.5 Using instruments to control our design

The instruments we will use will provide us a method with which to control our design and afford us an unprecedented view into the signals inside our device, in real-time. Below is a list of the instruments we will utilize for this next section:

Instrument	Role
Configurable Digital IO 	We will utilize the Configurable Digital IO component to control the rotation and scaling of our image on the TFT display.
Logic Analyzer 1k x 8 	We will use the Logic analyzer to monitor the signals coming out of the Wishbone Port IO component; specifically those being sent to the LEDs on the NanoBoard.
Frequency Generator 	We will use the Frequency Generator to create a time base for our Logic Analyzer.
Frequency Counter 	We will use the Frequency counter to watch the time base generated by our Frequency Generator and the signal being output to the LEDs.

10.6 Exercise 18 – Updating the Wishbone Port IO

Prior to adding the instruments, we need to make a modification to our design. Specifically, we need add a few additional IO ports to the Wishbone Port IO component. These IO ports will be used to interface the instruments to our design.

In this exercise we will update our Wishbone Port IO the Peripheral Interconnect components. These ports will be used by the host processor to read the Rotation and Scaling values from the Digital IO instrument. We will also include some input Flags that our software can trigger off of. Additional outputs will also be used to optionally display the rotation, scaling, and flag values as binary values on the Digital I/O component (thus giving us some visibility into just what our settings are).

1. Double-click the Configurable Wishbone Port I/O component U14 and hit the **Configure** button in the **Component Properties** dialog or right-click U14 in the schematic and select **Configure U14 (WB_PRTIO)**....
2. Change the **Port Count** value to **4**. This will give us 4 input ports, each 8 bits wide, and 4 output ports, also 8 bits wide.
3. We now need to update our Wishbone Interconnect component to be able to address each of these 4 different I/O ports. To do so double-click the Wishbone Interconnect component **U1** and hit the **Configure** button in the **Component Properties** dialog or right-click U1 in the schematic and select **Configure U1 (WB_INTERCON)**....
4. Highlight **GPIO** in the list of peripherals and select the **Edit Device** button at the bottom.
5. Edit the **Address Bus Width** to **2 Bits – Range = 4**. This will allow us to address each of the 4 I/O ports from the processor.
6. A new address pin will appear on the Wishbone Interconnect and the Wishbone Port I/O component (s2_ADR_O[1..0] and ADR_I[1..0] respectively). Using **Place»Bus**, wire these two pins to one another.
7. Additional pins have also been added to the left of the Wishbone Port IO component. Wire these pins and verify the wiring matches the image below.

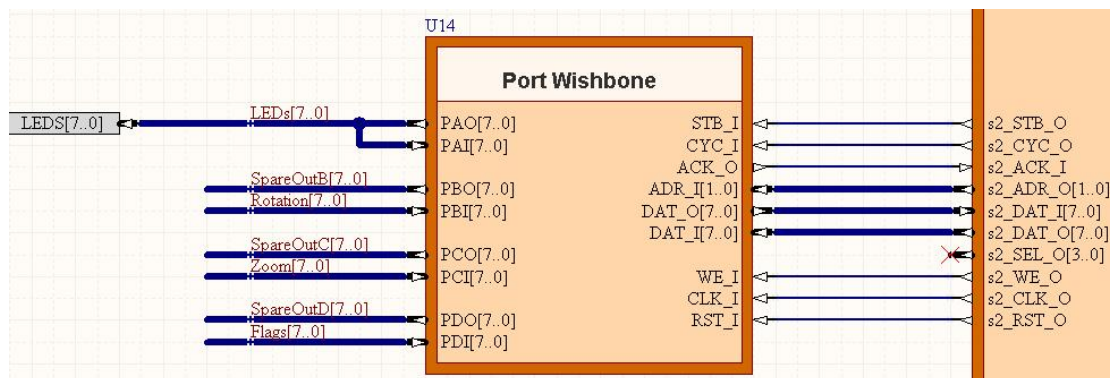


Figure 43. Placing and wiring the wishbone port IO.

Now with the Wishbone Port I/O and Wishbone Interconnect components updated we can begin the process of adding the Virtual Instruments to our design. In the following exercises we will add each of the instruments we intend to use.

10.7 Exercise 19 – Adding the Virtual Instruments

1. Go to the **Libraries** panel and place the component **DIGITAL_IO**. Designate this component **U11**.
2. Right click this component and select **Configure U11 (DIGITAL_IO)...** Configure the component as it appears in Figure 44.

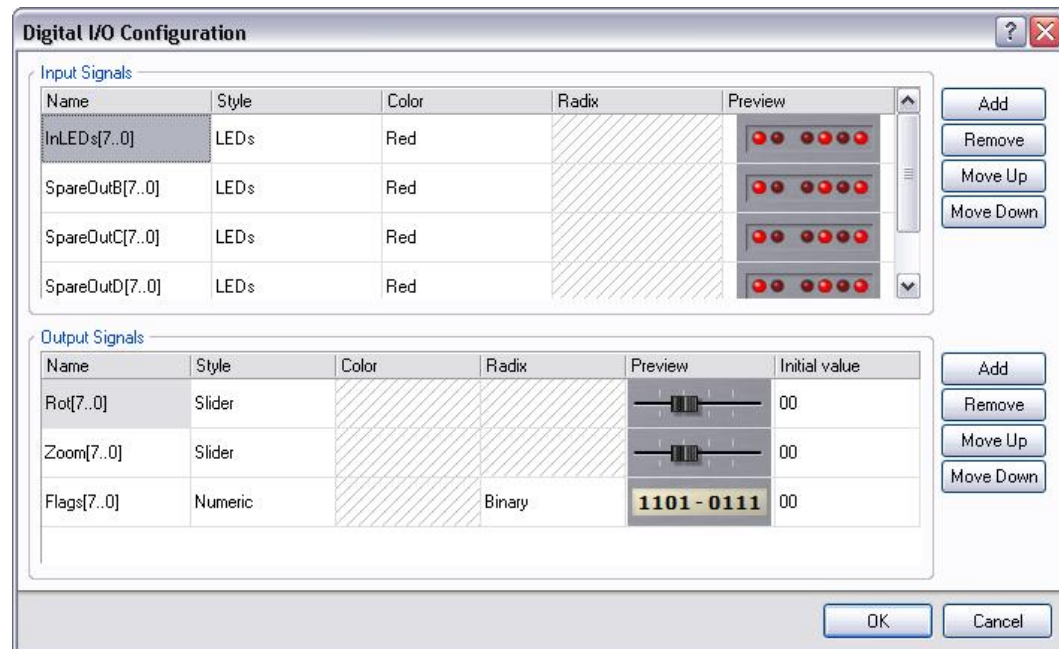


Figure 44. Digital IO component configured for the design

3. Wire the component as indicated in Figure 45. This will tie together the signals on our Wishbone Port IO device and the Configurable Digital IO instrument.

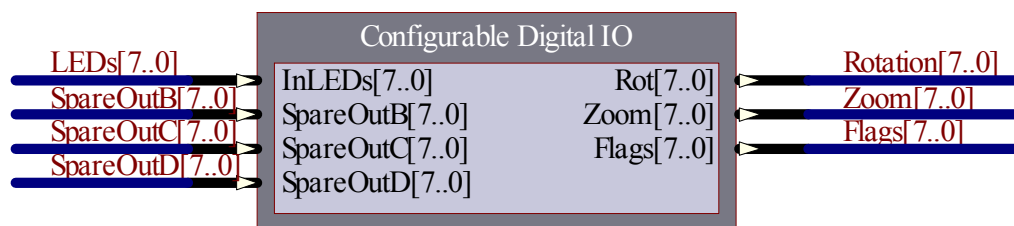


Figure 45. Wiring for the configurable digital IO

4. Place the clock generator component CLKGEN, designating this component U19
5. Place the frequency counter component FRQCNT and designate this component U18
6. Wire both components as they appear in Figure 46.

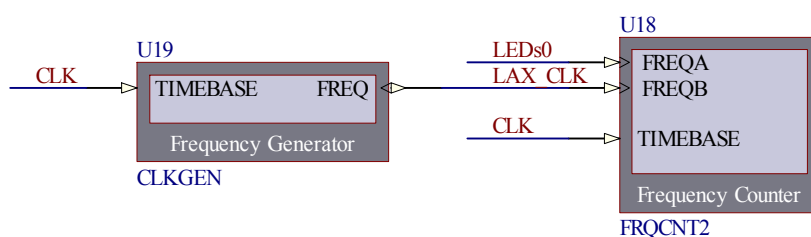


Figure 45. Wiring for the frequency generator and frequency counter instruments

7. Place the logic analyzer component LAX, designating this component U42.
8. Configure the LAX with one signal set *Measure*, and a single signal *LEDS[7..0]* as seen in Figure 46.

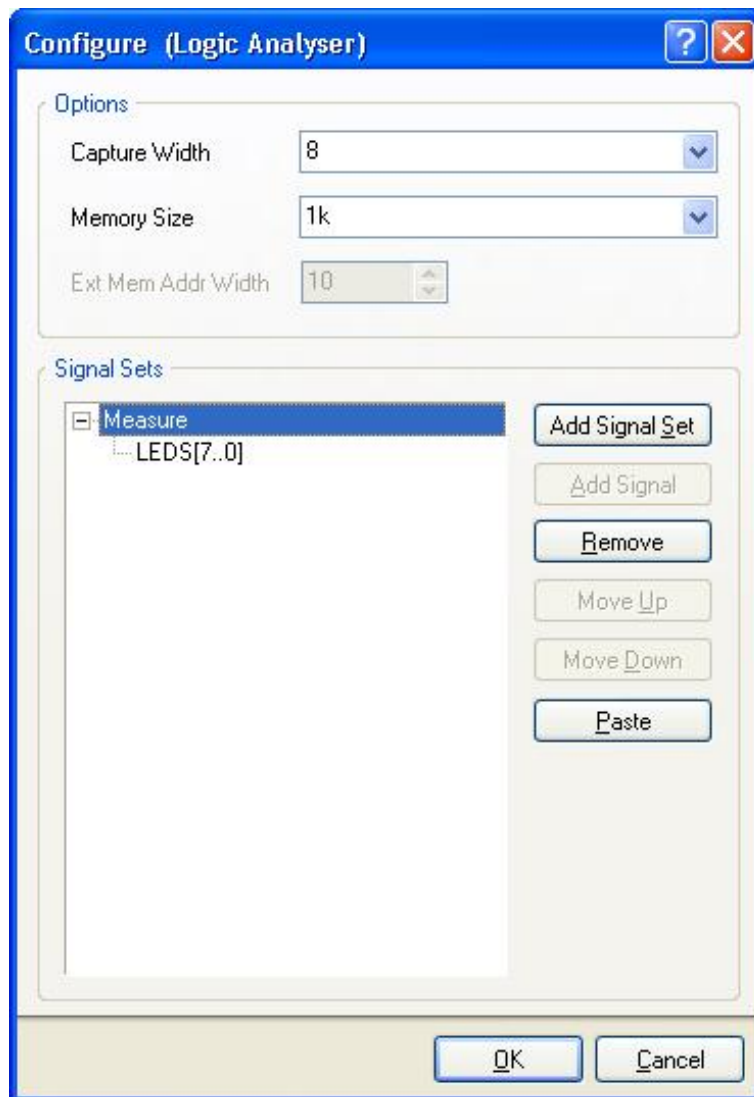


Figure 46. LAX configured to capture input from the LEDS

9. Wire the LAX as it appears in Figure 47.

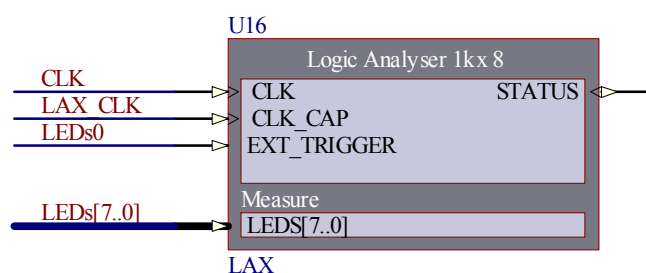


Figure 47. Wiring for the configurable LAX

10.8 Exercise 20 - Rebuilding the project with instruments

1. Make sure your Desktop NanoBoard is connected to your PC and powered on.
2. Select **View >> Devices View** or click on the **Devices View** icon  in the toolbar.
3. Ensure that the **Live** checkbox is checked. You should see a picture of the Desktop NanoBoard in the upper region of the display and an icon of the Spartan3 FPGA in the middle region.
4. In the drop down list just below the Spartan3 icon, ensure that the **CHC_image_rotation / NBD2DSK01_07_DB30_04** project / configuration pair is selected.
5. Locate the **Compile, Synthesize, Build, Program FPGA** buttons running left to right just below the Desktop NanoBoard icon. Click once on the words **Program FPGA** to begin the build process.

10.9 Updating our embedded project to use the instruments

To make use of the virtual instruments we will require some changes to the application code in our embedded project. This includes a piece of software that makes zooming and scaling dependent on the two sliders we've added using our Configurable Digital IO Instrument. In this section we will modify our software and embedded project to get the video working with our instruments.

10.10 Exercise 21 – Adding Additional Source Files

To further simplify our embedded project we will do some simple partitioning. This makes the source files easier to read and manage and makes our files far more reusable.

1. Right click the embedded project **EmbeddedProject.PrjEmb** and select **Add New to Project**. Add a new **H File**.
2. Add the contents of the snippet `Spinning_Video_Defines_Header` to this source file.
3. Save the file as **defines.h**.
4. Right click the embedded project **EmbeddedProject.PrjEmb** and select **Add New to Project**. Add a new **H File**.
5. Add the contents of the snippet `Spinning_Video_DSF_Header` to this source file.
6. Save the file as **dsf.h**.
7. Right click the embedded project **EmbeddedProject.PrjEmb** and select **Add New to Project**. Add a new **C File**.
8. Add the contents of the snippet `Spinning_Video_DSF_SourceFile` to this source file.
9. Save the file as **dsf.c**.

10.11 Exercise 22 – Updating main.c

In this exercise we will update main.c to include the application code necessary to respond to our inputs and rotate and scale our video.

1. Open the main.c source file in the project **EmbeddedProject.PrjEmb**
2. Select and delete all of the source code located in this file.
3. Add the contents of the snippet `Spinning_Video_main_SourceFile` to this source file.
4. Save the file as **main.c**
5. Save the embedded project **EmbeddedProject.PrjEmb** by right clicking the project document in the **Projects** panel and selecting **Save Project**.

- Return to `main.c` and press the **Compile and Download** button  in the toolbar at the top of the screen to recompile the software and update your changes to the processor running on the FPGA.
- With the aid of your instructor, have a look at the source code. The bulk of the capability rests in the `main.C` file.

10.12 Exercise 23 – Controlling our design with instruments

In this exercise we will familiarize ourselves with the interface to the Digital IO Virtual Instrument.

- Switch to the Devices View under **View»Devices Views** or by hitting the  button in the toolbar at the top of the screen. Select and delete all of the source code located in this file.
- The soft nexus JTAG chain should display all of the instruments we've added to our design, as well as the host processor.

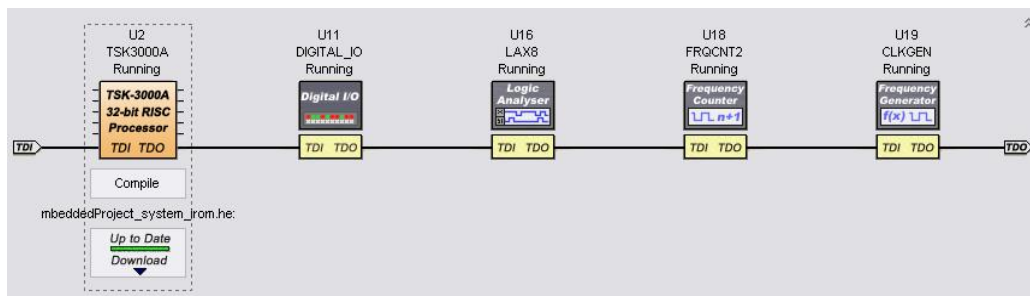


Figure 48. Soft devices chain shows the instruments in our design

- Right click the U11 DIGITAL_IO and select **Instrument** to launch the Instrument Rack for this soft device.

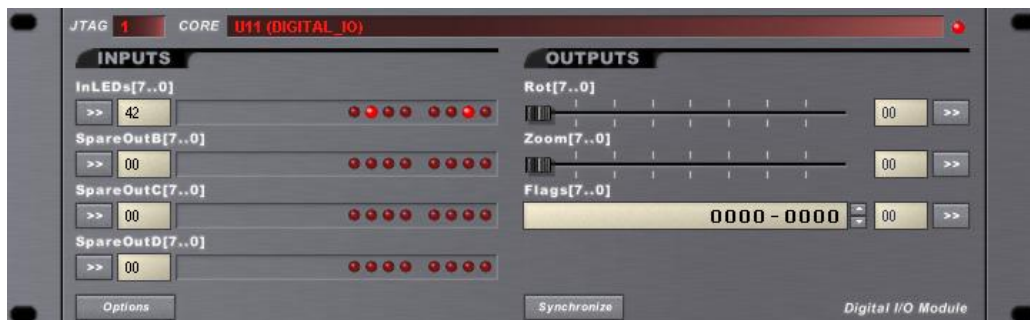
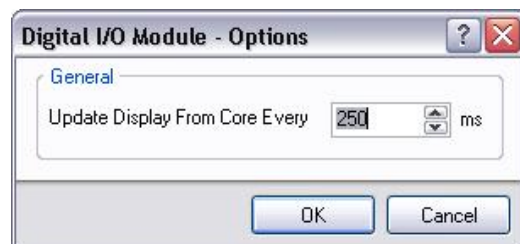


Figure 49. Instrument rack for the configurable Digital IO

The **INPUTS** region of the instrument panel allows you to monitor the digital signals that have been wired to the input channels of the DIGITAL_IO instrument.



- Click on the **Options** button, at the bottom left of the panel, to access the *Digital I/O Module – Options* dialog. The display for each input is refreshed (updated) in accordance with the value specified in the **Update Display From Core Every** field of this dialog.
- Move the slider `Rot[7..0]` to the right and observe the image rotating on the TFT.
- Move the slider `Zoom[7..0]` to the right and observe the image scaling on the TFT.

7. Recall in our source code we have created 2 conditions for Flags[7..0]. The first condition was in the form of an If...Else statement that read:

```
if (Flags & 0x01)
    alpha += Rotation;
else
    alpha = Rotation << 1;
alpha = alpha >= 360 ? 0 : alpha;
```

If bit 0 of the array Flags[7..0] is set, the image will continue to rotate (the variable `Rotation` refers to the B input PBI[7..0] of our Wishbone Port IO component).

8. Set the right-most bit (the LSB) of Flags[7..0] to see the image on the TFT change to a state in which it is rotating continuously.
9. Recall in our source code that we had a second condition controlled by Flags[7..0], this time controlling our scaling of the image:

```
if (Flags & 0x02)
{
    scale += delta * Zoom;
    if (scale > 1024)
        delta = - 1;
    if (scale < 256)
        delta = 1;
}
else
    scale = 256 + Zoom * 3;
```

Under this condition, the second bit of Flags[7..0] will cause the scaling to continue to both increase and decrease.

10. Set the second bit of Flags[7..0] and notice the image changes to both increase and decrease in a continuous motion.
11. Time permitting, experiment with some of the other instruments in the design.

11 Real-time Debugging of a Processor

Altium Designer includes a range of tools to debug our soft processor. This includes a more traditional simulation environment; as well as support for debugging real software, on real hardware, in real-time. It is this capability that coupled with the Virtual Instruments, sets Altium Designer apart.

In this section we will discuss some of the software debugging tools available to you in Altium Designer.

11.1 On-Chip Debugging

Each of the microprocessor / microcontroller cores supplied with Altium Designer include support for **On Chip Debugging (OCD)**. The 8-bit microcontrollers will utilize a separate schematic symbol with an '**_D**' appended to the schematic symbol name to distinguish the OCD and non-OCD versions of the component. Whereas the 32-bit processors contain an option in their respective configuration dialogs to include **On Chip Debugging** resources. The OCD system is necessary if you wish to make use of Altium Designer's live debugging features. One notable difference between the two variations of microprocessors / microcontrollers is the existence of program memory *write* resources on the OCD version. This is necessary since the debugger requires write access to program memory so that it can place breakpoints in program memory when breaking or stepping through code. Thus when using the OCD version, program memory *must* be comprised of read/write memory. In general it is advisable to use the OCD version during development and switch back to the non-OCD version for production.

11.2 A word about simulation

In general, it is desirable to interact with real hardware when debugging rather than performing simulations only. The presence of the NanoBoard means that most designs can be debugged in real hardware, i.e. LiveDesign, however there may be the odd occasion when it is necessary to simulate a design.

The Altium Designer development environment supports full simulation based debugging as well as LiveDesign using real hardware; the interface is identical. Switching between LiveDesign and Simulation is achieved by changing the debugger mode via the **Debug** toolbar available when editing an embedded project source document, or by right clicking the Embedded Project in the **Projects** panel and selecting either **Debug <processor designator>** or **Simulate**.

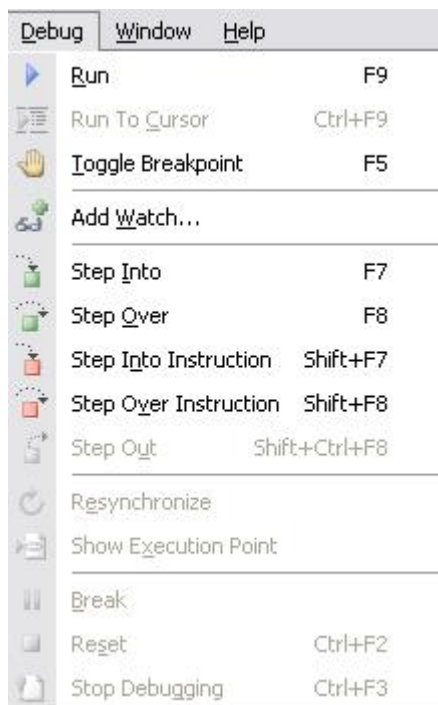
For our discussions we shall focus on the LiveDesign environment, as this is usually the most desirable mode of debugging however all that is mentioned here is applicable to the simulation environment also.

The easiest way to initiate a debugging / simulation session is to right-click the embedded project in the Projects panel and selecting Debug or Simulate. Of course, the Debug option is only available if a NanoBoard or target platform is present and powered up.

11.2.1 The Debug menu

Debug commands are available whenever an embedded source file is open in the main window. Debug commands can be accessed via the **Debug** toolbar:





...Or from the **Debug** menu:

Run (F9): Run the current embedded project in debug mode. After launching the command, execution of the code in the embedded software program will begin. If any enabled breakpoints have been setup, code execution will halt at these points, if encountered

Run to Cursor (Ctrl+F9): Execute the embedded code up to the line containing the cursor.

Toggle Breakpoint (F5): This command is used to toggle an enabled breakpoint for the current line. After launching the command, the current line will have an enabled breakpoint added to it or removed from it, depending on whether or not an enabled breakpoint currently exists for that line.

Enabled breakpoints are indicated in the code by a red highlight over the breakpoint line and a red circle with a cross in the margin.

Disabled breakpoints are indicated in the code by a green highlight over the breakpoint line and a green circle with a cross in the margin. A disabled breakpoint remains defined but will not cause running code to halt when encountered.

If a disabled breakpoint exists for the line and this command is used, the breakpoint will be removed.

You can view a list of all breakpoints that have currently been defined in all open embedded project source code files, in the **Breakpoints** panel.

Add Watch: This command enables you to define watch expressions for the current embedded source code document. A watch expression can be a single variable or an expression containing one or more variables and allows you to view the value of the expression as you step through the code. Basic mathematical operations are supported (e.g. $a+b$, $a*b$, $c+(b-a)$).

Step Into (F7): Use this command to execute each line of the current embedded source code sequentially, including the individual lines of code contained within any procedures/functions that are called. The next executable line of code is highlighted in blue and is indicated by a blue circle with an arrow in the margin.

Step Over (F8): The same as the Step Into command except procedure/function calls are treated as a single line of code and executed as a single step.

Step Into Instruction (Shift+F7): This command is used to execute each individual instruction at the assembly code level, in turn, including the instructions contained within any functions that are called.

When the source code document is an .asm file, the next executable instruction is highlighted in blue and is indicated by a blue circle with an arrow in the margin. This command and the **Step Into Source** command will behave in the same way.

When the source code is a high level language (.c file), use of this command should ideally be made from within one of the two disassembly views for the code - either the extended mixed source-disassembly view (accessed by clicking the **Show Disassembly** button on the debug toolbar in the source code view), or the pure disassembly view (accessed by clicking the **Toggle Source Code** button on the disassembly standard toolbar, from within the mixed source-disassembly view).

In both mixed and pure disassembly views, the next executable instruction is highlighted in dark blue and is indicated by a dark blue circle with an arrow in the margin.

Step Over Instruction (Shift+F8): The same as the **Step Into Instruction** command except procedure/function calls are treated as a single line of code and executed as a single step.

Step Out: This command is used to step out of the current function within the embedded source code. The remaining executable code within the current function will be executed and the execution will be passed onto the next sequential line of code after the function's calling statement.

Show Disassembly: Open an intermixed source and disassembly view for the current embedded software project. A new view will open as the active view in the main design window. This view shows a mixture of disassembled instructions and source (C) code. The source for all source code files in the current embedded project will be displayed.

In this intermixed disassembly and source view, the next executable source line is highlighted in blue and is indicated by a blue circle with an arrow in the margin. The next executable disassembled instruction is highlighted in dark blue and is indicated by a dark blue circle with an arrow in the margin.

Resynchronize: Use this command to synchronize the debugger execution point with the external hardware.

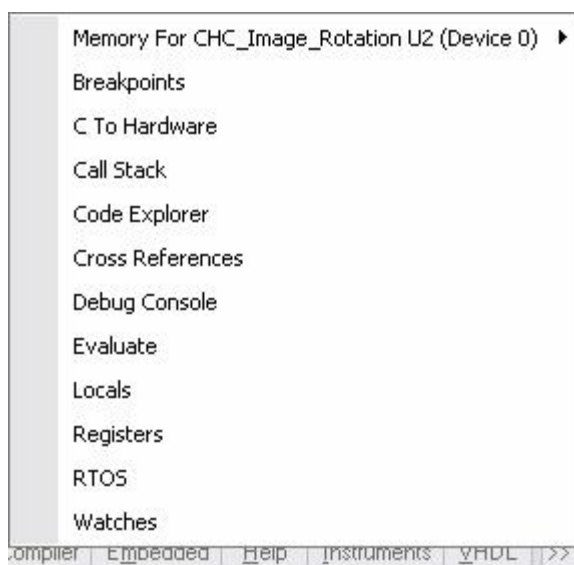
Show Execution Point: Position the text cursor at the start of the next line of code to be executed. If the next executable line of code is outside of the visible area of the main display window, the document will be panned to bring it into view.

Break: Halt an executing processor at the next executable line of source code.

Reset (Ctrl+F2): Reset the executing processor currently being debugged, at any stage when stepping through code or after a breakpoint has been encountered, and return the current execution point back to the first line of executable code.

Stop Debugging (Ctrl+F3): Terminate the current debugging session.

11.3 Embedded control panels

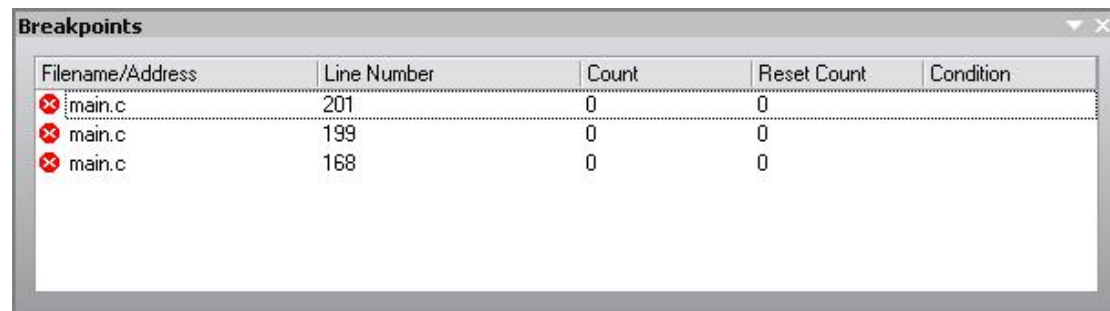


Clicking on the **Embedded** button of the workspace panels will open the list of embedded control panels. Alternatively use the menu commands by selecting **View » Workspace Panels » Embedded**.

Selecting **F1** whilst an item within a panel has the focus will bring up extensive help on the panel's operation.

Figure 48. Debugger panels

11.3.1 Breakpoints



Filename/Address	Line Number	Count	Reset Count	Condition
main.c	201	0	0	
main.c	199	0	0	
main.c	168	0	0	

Figure 49. The breakpoints panel

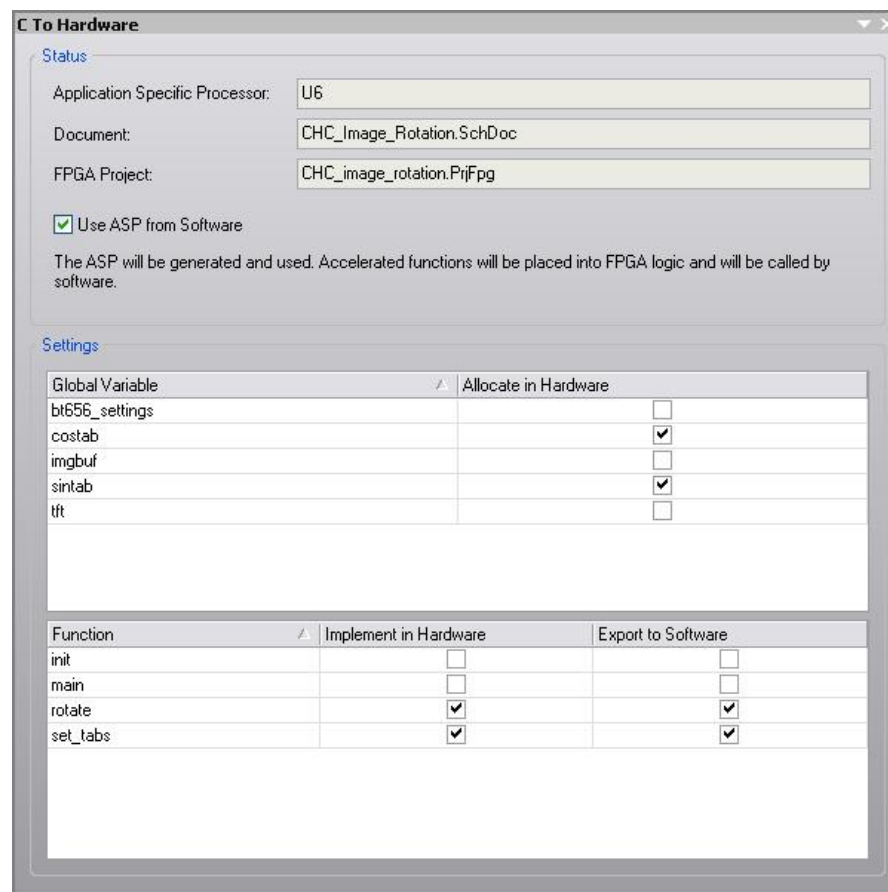
The **Breakpoints** panel provides information on all breakpoints that are currently defined - either within the open source code documents for any embedded project or in one of the memory address spaces associated with a processor whose embedded code is currently being debugged.

The following two types of breakpoint can be defined:

- **Source code breakpoint** - applied to a particular line of code in the source document of an embedded software project
- **Memory address breakpoint** - specified for a particular address in a memory space associated with the processor on which the code is running.

Any breakpoints that are currently defined in any open embedded project source code documents (*.C, *.asm), or processor memory spaces will be listed in the panel.

11.3.2 C to Hardware



Status

Application Specific Processor: U6

Document: CHC_Image_Rotation.SchDoc

FPGA Project: CHC_image_rotation.PrjFpg

☒ Use ASP from Software

The ASP will be generated and used. Accelerated functions will be placed into FPGA logic and will be called by software.

Settings

Global Variable	Allocate in Hardware
bt656_settings	☐
costab	☒
imgbuf	☐
sintab	☒
tft	☐

Function	Implement in Hardware	Export to Software
init	☐	☐
main	☐	☐
rotate	☒	☒
set_tabs	☒	☒

Figure 50. C to Hardware panel.

The C to Hardware panel provides a control panel for Altium Designer's Application Specific Processor discussed later in this course. From this panel Global Variables and Functions can be 'off-loaded' to hardware provided they meet the criteria required to instantiate the particular variable or function as logic. This capability is detailed in the document GU0122 C-to-Hardware Compiler User Manual.pdf

11.3.3 Call Stack Panel

The **Call Stack** panel provides information about which line of source code the debugger is next going to execute and which function that line of code resides in. If a sequence of function calls have been made to reach the current line of code, these calls will be displayed as a 'call stack'.

As you step-debug your source code, the panel will show the current function you are in, the name of the active source document and the line of source code that the debugger has reached (i.e. the next line of executable code).

When a call to another function is encountered, the panel will show that function, along with the values of any parameters passed into it and the next executable line of code within that function. As the function is called from within a function, the original function is still displayed, showing the line of code that will be executed when program execution passes back into it.

A 'call stack' is effectively set up, showing the relationship between descendant function calls. The original function appears at the bottom of the stack, with each subsequently called function above it, right up to the currently entered function on the top of the stack. The current function is distinguished by a red arrow to its immediate left.

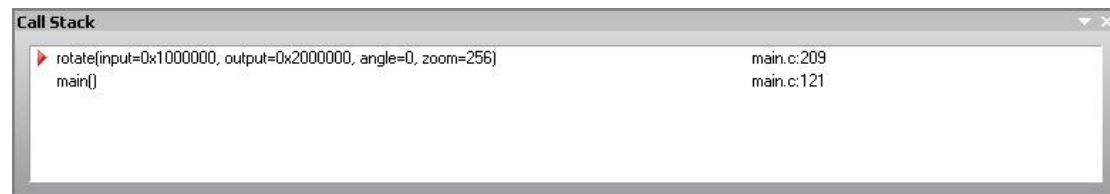


Figure 51. The Call Stack panel.

11.3.4 Code Explorer

The **Code Explorer** panel provides a visual summary of all identifiers (Define statements, Variables, Routines (functions) or Labels) that are used in the active source document (*.C, *.asm, *.h) for an embedded software project (*.PrjEmb).

The information that appears in the Code Explorer panel depends on the type of source document currently active in the design editor window.

Double-clicking on an entry in the panel (or selecting it and pressing ENTER) will jump to the corresponding area of code in the design editor window, placing the text cursor to the immediate left of the identifier.

Direct filtering is available, allowing you to quickly jump to an entry by directly typing within the panel.

To use this feature, simply click within the panel and type the first letter of the entry you

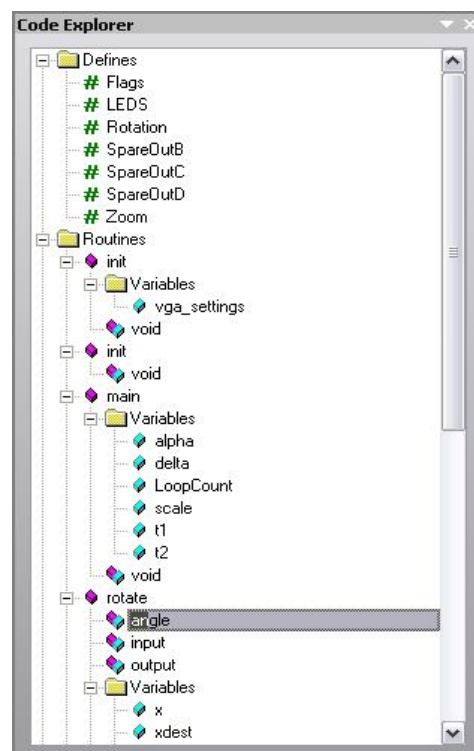


Figure 52. The code explorer panel

wish to jump to. The first entry in the panel starting with the letter you type will become selected and the letter will be highlighted.

11.3.5 Cross References

The **Cross References** panel enables you to quickly locate all occurrences of a variable, function or procedure, within the source code documents of an embedded project.

Information in the **Cross References** panel will only appear after right-clicking on an identifier in the source code (variable, function or procedure) and choosing the Show Cross References command from the subsequent pop-up menu that appears.

Note: Cross referencing must be set up for the source code documents for this feature to be available. Enable **Create Cross-Reference Info** under **Tools»Embedded Preferences»Embedded System»Libraries**. Refer to the document TR0104 Altium Designer Panels Reference.PDF for complete details on setting up your preferences to support cross-referencing.

The upper region of the panel lists instances of the selected variable, function or procedure name that have been found within the scope of the search. As you click on an instance, the lower region of the panel will show where in the source code that instance exists.

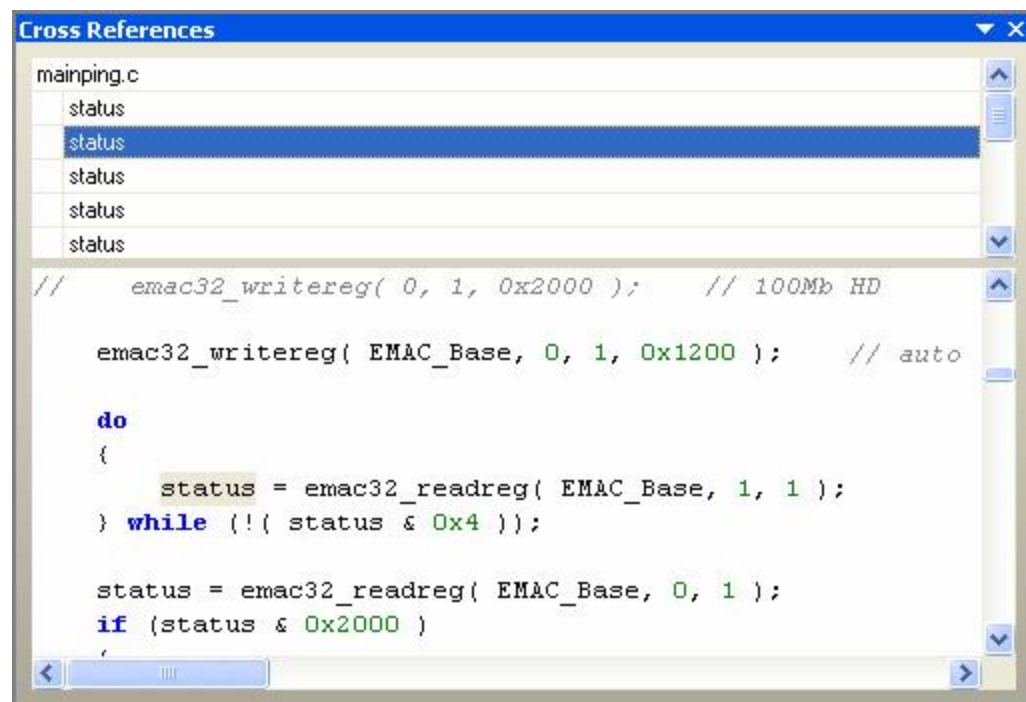


Figure 53. The cross references panel

11.3.6 Debug Console

The **Debug Console** panel provides a history of the current debug session, in terms of the commands issued and the lines of code executed. The panel also offers command-line debugging of the embedded code.

As you debug the source code for the active embedded project, the panel will 'keep track' of the debug commands as

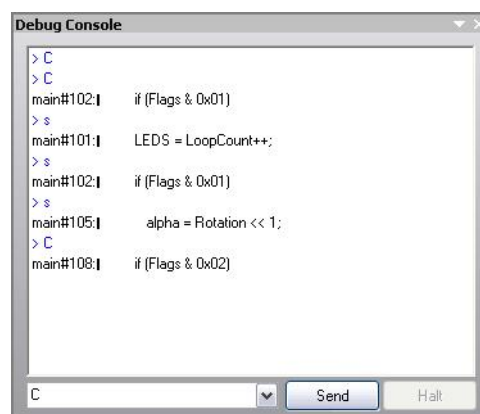


Figure 54. The debug panel.

you issue them and the current execution point. For example, if you issue a **Step Into** command from the

Debug menu or toolbar, the panel will show an entry that reflects this command (>s) followed by an entry that reflects the next line of code to be executed.

The controls at the bottom of the panel allow you to drive the debug session from within the panel. Simply enter the syntax for the command you wish to issue to the Debugger and either click the **Send** button or press **ENTER**.

The following table contains example command-line entries that correspond to common debugging commands available from the **Debug** menu or toolbar:

Panel Command-line Entry	Corresponding Debug Command on Debug Menu	Corresponding entry in Debug toolbar
C	Run	
C n (where n = line number)	Run to Cursor	
s	Step Into	
S	Step Over	
si	Step Into Instruction	
Si	Step Over Instruction	
.dxc resync	Resynchronize	
rst	Reset	

In addition, the **HALT** button at the bottom right of the panel behaves the same way as the **Break** command available on the **Debug** menu and toolbar.

11.3.7 Evaluate Panel

The **Evaluate** panel is used to quickly determine the current value of a variable or expression in the active high-level source code document (*.C) currently being debugged.

Use the field at the top-left of the panel to enter the variable or expression you wish to evaluate and either click the **Evaluate** button or press **ENTER**. The variable/expression string will appear in the main region of the panel, with the result of the evaluation appearing next to it, under the **Value** column.

The variable/expression is not updated with respect to its value as you step-debug the code. You can, however, repeatedly use the **Evaluate** button to update the value for the entry.

If you wish to keep a constant watch on the variable/expression, where the value is updated as the code executes, use the **Add Watch** button to add the variable/expression as a new watch entry to the **Watches** panel.

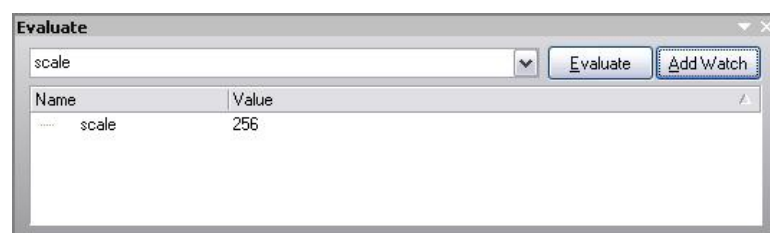


Figure 55. The evaluate panel

11.3.8 Locals Panel

The **Locals** panel allows you to interrogate the values of variables local to the function that the debugger has currently entered. You can also change the values of variables on-the-fly as you debug.

As you step-debug your source code, the panel will display all local variables and passed parameters as they are encountered for the current function. When entering a parametric function, the entries will appear in the panel after execution of the calling statement.

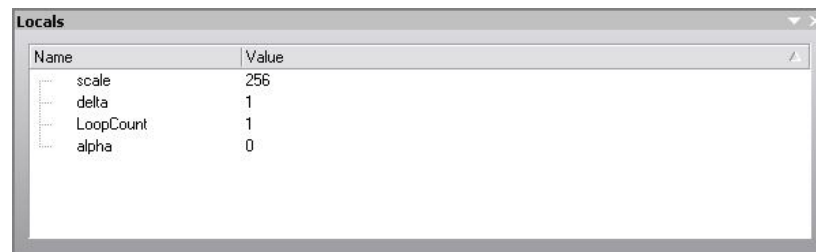


Figure 56. The locals panel

11.3.9 Registers Panel

The **Registers** panel allows you to interrogate the content of registers within the core processor currently being debugged. You can also change the values of registers on-the-fly as you debug.

For each register, the following data fields are available:

- **Register** - the name of the register
- **Width** - the size of the register (e.g. 8-bit, 16-bit, 32-bit)
- **Decimal** - the value currently stored in the register, presented in decimal format (e.g. 30)
- **Hexadecimal** - the value currently stored in the register, presented in hexadecimal format (e.g. 1E)
- **Binary** - the value currently stored in the register, presented in binary format (e.g. 0001-1110)
- **Char** - the value currently stored in the register, presented in character format (e.g. decimal 87 would be character W)

The **Register**, **Width** and **Char** fields are non-editable. The number format fields allow for changing of register values as you debug.

Registers

Current CPU status: 31 ack vie ite itr umo ieo ump iep umc iec

Register	Width	Decimal	Hexadecimal	Binary	Char
0 Status	32	63488	0000F800	0000-0000-0000-0000-1111...	
1 IEnable	32	0	00000000	0000-0000-0000-0000-0000...	
2 IPending	32	0	00000000	0000-0000-0000-0000-0000...	
3 IMode	32	0	00000000	0000-0000-0000-0000-0000...	
4 TBLO	32	3542568087	D3274C97	1101-0011-0010-0111-0100...	.L.
5 TBHI	32	7	00000007	0000-0000-0000-0000-0000...	
6 PIT	32	4294967295	FFFFFFF7	1111-1111-1111-1111-1111...	
7 ER	32	65792	00010100	0000-0000-0000-0001-0000...	
8 EB	32	256	00000100	0000-0000-0000-0000-0000...	
9 PC	32	2348	0000092C	0000-0000-0000-0000-0000...	
10 HI	32	0	00000000	0000-0000-0000-0000-0000...	
11 LD	32	33546240	01FFE000	0000-0001-1111-1111-1110...	
12 \$sp \$31	32	21332	00005354	0000-0000-0000-0000-0101...	..ST
13 \$0	32	0	00000000	0000-0000-0000-0000-0000...	
14 \$1 at	32	65536	00010000	0000-0000-0000-0001-0000...	
15 \$2 \$v0	32	1	00000001	0000-0000-0000-0000-0000...	
16 \$3 \$v1	32	0	00000000	0000-0000-0000-0000-0000...	

Figure 57. The registers panel

11.3.10 Watches Panel

The **Watches** panel enables you to create and display a list of watch expressions, allowing you to keep track of variable/expression values as you single-step debug the source code of an embedded software project.

The panel lists, for each watch that is added, the **Expression** (what to watch for) and the **Value** (the evaluation of the expression at the current line in the source code).

A small icon to the left of the watch expression is used to indicate the status of the watch:

-  - Source code execution has entered a function or procedure where the watch expression is valid
-  - Source code execution has entered a function or procedure where the watch expression is invalid. This state will also occur if the watch expression is not valid.
-  - Watch is disabled

Watches

Expression	Value
 y	29
 xdest	-62
 x	-62
 zoom	256

Figure 58. The watches panel

11.4 Instrument Rack – Nexus Debugger

In addition to the traditional source-code level debug tools available, Altium Designer also includes an **Instrument Rack** for microprocessors / microcontrollers that gives you an unparalleled view of the processor's registers, and the opcodes and instructions executing on the current processor. This dialog will also allow you to pause, run, reset, and step the processor.



Figure 59. Instrument rack for soft devices

This instrument rack is accessed by **right-clicking** the processor in the **Devices** view and selecting **Nexus Debugger**.

Clicking on the **Nexus Debugger** button will access the corresponding debug panel for the type of processor you are using. This panel contains a greater array of debug controls, providing access to the internal registers, memory spaces and also shows a disassembly view of the embedded source code as you debug.

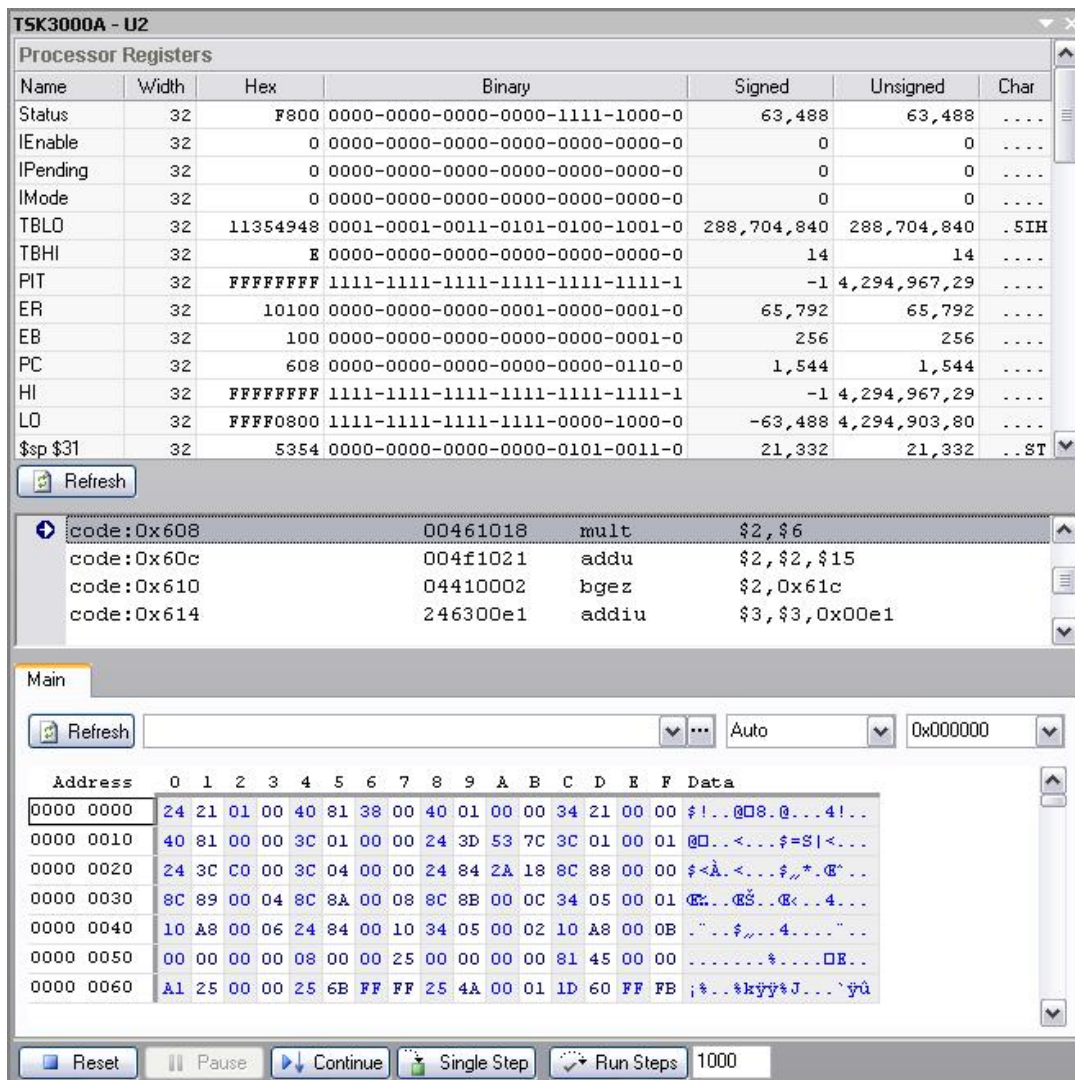


Figure 60. Nexus debugger panel

The various memory panels available for a processor enable you to interrogate the different memory spaces associated to that processor, concurrently.

The behavior of each of the available memory panels is identical, regardless of the memory space represented:

- address space typically starts at 00000000h when displayed in the panel, regardless of where the memory physically resides in the processor's memory map
- memory space is divided into rows of 16 cells, with each cell representing an address in memory space and the value contained therein, in hex format. The **Data** column reflects the value at each address (in the associated row) in ASCII format
- a field is included at the top of each panel for filtering, allowing you to jump to row containing the specific address of interest within the represented memory space
- a refresh button at the top left of the panel enables you to manually refresh/update the contents of the memory space

11.4.1 Changing the Contents of Memory

To change the contents of an address in memory, simply click on the desired cell and type the new hexadecimal value as required. If the memory is writable, it will automatically be refreshed with the entered value. If it is read-only, the new value will be discarded and the memory location will remain unchanged.

Additional editing options including options to Fill a memory with zeros, ones, or a hex value 0xFF, as well as options to Save the memory contents to a file or Load memory contents from a file. Simply right-click a memory in the dialog to view a full list of options.

11.4.2 Defining Breakpoints in Memory

Memory address breakpoints can be added directly to addresses from within the relevant panel for the required memory space. To add a breakpoint at a single address, simply right-click over that address in memory and choose when the breakpoint is to be applied (on read, on write, on read or write).

12 C to Hardware

In this section we will introduce you to the C to Hardware capability found in Altium Designer. The C-to-Hardware Compiler (CHC) compiler resembles a normal toolset (compiler, assembler, linker, and locator) but instead of producing a software file, it produces a hardware file that can be loaded onto an FPGA.

The C-to-Hardware Compiler accepts standard untimed ISO-C source code as input and produces a synthesizable hardware file. The synthesis tools of Altium Designer translate this hardware file into an electronic circuit which can be loaded onto an FPGA along with the rest of the design and the software.

In Altium Designer, the C-to-Hardware Compiler is used in combination with a traditional embedded compiler. This means you can write an 'ordinary' C program for a processor core. Using Altium Designer, you can mark which functions should be compiled to hardware. Your FPGA design then needs an Application Specific Processor (ASP) to hold these compiled hardware functions.

When compiling and synthesizing your project (embedded software and hardware), either the regular C compiler or the C-to-Hardware compiler is invoked depending on whether you marked functions as hardware or not.

The final result, after having compiled the application and having loaded both software and hardware onto an FPGA, is a system where the software and the synthesized hardware functions form the implementation of your original C program. The software part 'calls' hardware functions that perform their task far more efficiently than if they were compiled to software.

12.1 What are the benefits of the C-to-Hardware Compiler?

Virtually all C programs (or functions) can be converted to an electronic circuit by the C-to-Hardware Compiler. However, the characteristics of the program determine whether the C-to-Hardware Compiler can create an efficient hardware component or whether it is better to execute the program on a processor core. The C-to-Hardware Compiler can only create a small and fast electronic circuit if the C source code is *parallelizable*. In such a case the hardware executes many operations in parallel whereas a processor core would fetch and execute instructions sequentially.

Graphics, signal processing and encryption algorithms translate very well into hardware and performance improves by orders of magnitude. For these types of algorithms FPGA implementations outperform high-end DSP and RISC processor cores.

So the main benefit of the C-to-Hardware Compiler is, that it lets you design hardware modules to perform specific tasks by simply programming them in C. Normally this would be a complex and time-consuming job that would have to be performed by specialized hardware designers.

12.2 Using the CHC Compiler

For a regular embedded project, the CHC compiler will not be invoked. In this situation only the FPGA design is synthesized and loaded onto an FPGA together with a ready compiled embedded project.

The C-to-Hardware Compiler is actually a complete toolset, including a compiler, assembler and linker, conceptually very similar to a regular C toolset.

The CHC compiler is invoked when the FPGA design gives cause to do so:

- the FPGA design contains one or more Application Specific Processor (ASPs) components, which are able to hold the hardware equivalent of a C software function
- the configuration of the ASP-component describes that it contains at least one hardware compiled C function

When building the project, based on the configuration settings of the ASPs on the schematic, a list of function qualifiers is generated that mark the functions to be compiled to hardware. This function qualifier file is submitted both to the regular embedded toolset and to the CHC toolset. The embedded toolset now "knows" which functions should be compiled to hardware. It does not compile these functions, but generates a wrapper with just enough code to call the hardware function, to pass any variables, and to receive the return value. The result is an absolute ELF object file.

The same C sources and function qualifier file are submitted to the CHC toolset. Based on the function qualifier file, the CHC toolset compiles and assembles the marked C functions to hardware. The result is an absolute ELF object file which contains the hardware functions.

The ELF file with the hardware functions will be translated into VHDL or Verilog by the HDL generator. Then it is synthesized, along with the rest of the FPGA design, resulting in a BIT file which can be loaded onto the FPGA. The ELF file with the software functions is already in its final state and will be loaded into the processor soft-core on the FPGA.

12.3 Implementing C to Hardware in our Design

In the following section we will prove that seeing, truly is believing. Recall the performance of the rotating video. Using the slider we saw the video scale and rotate on the TFT. As we complete the following series' of exercises we will see the performance of these two operations increase considerably.

12.4 Exercise 24 – Making Room for C to Hardware

In this exercise we will add an additional interface to our peripheral Wishbone Interconnect component U1. This provides us a means for the processor to access the functions that have been placed in hardware.

1. Right click U1 in the schematic and select Configure U1 (WB_INTERCON)...
2. Select the Add Device button and configure this device as seen in the figure below

Device Properties

Slave Name and Type
 Identifier: ASP
 Type: ASP
 Choose the identifier and type of the device.

Address Base
 00600000
 This is the base address of the device. For peripheral in the I/O space this is a 24-bit hexadecimal number and for memory devices this a 32-bit number.

Address Bus Mode
 Word Addressing - ADR_O(0) <= ADR_I(1 or 2)
 Byte addressing will result in master ADR_I(0) will be translated to slave ADR_O(0).
 Word addressing will be dependent on the data width: For 32-bit wide devices master ADR_I(2) will be translated to slave ADR_O(0) providing 4-byte words at each address. For 16-bit wide devices master ADR_I(1) will be translated to slave ADR_O(0) providing 2-byte words at each address. For 8-bit wide devices this mode is the same as byte addressing.

Decode Addressing
 4
 Controls how many of the address bits are decoded to select the peripheral. Decoders are generated automatically. For example, a value of 8 would mean that ADR(31 DownTo 24) (or ADR(23 DownTo 16) for 24 bit) is compared against the upper 8-bits of the Decode Address to select the peripheral. The smaller the number, the lower the hardware overhead but the less the number of different devices can be used.

Address Bus Width
 6 Bits - Range = 64
 This represents the number of address bits that are required to drive the slave.

Data Bus Width
 32-bit
 The width of the data bus on the slave device.

Graphical Attributes
 4
 Controls the extra space after the slave's bank of pins.

Used Interrupts
 ...
 Any processor interrupt lines that are not used by the slaves will be available as a single Spare_INT_I pin on the WB_INTERCON

OK Cancel

Figure 60. Adding the ASP to the wishbone interconnect

3. Click OK to close the Device Properties dialog.
4. Click OK to close the Configure U1 (Wishbone_Intercon) dialog.
5. Save your work.

12.5 Exercise 25 – Adding the Application Specific Processor

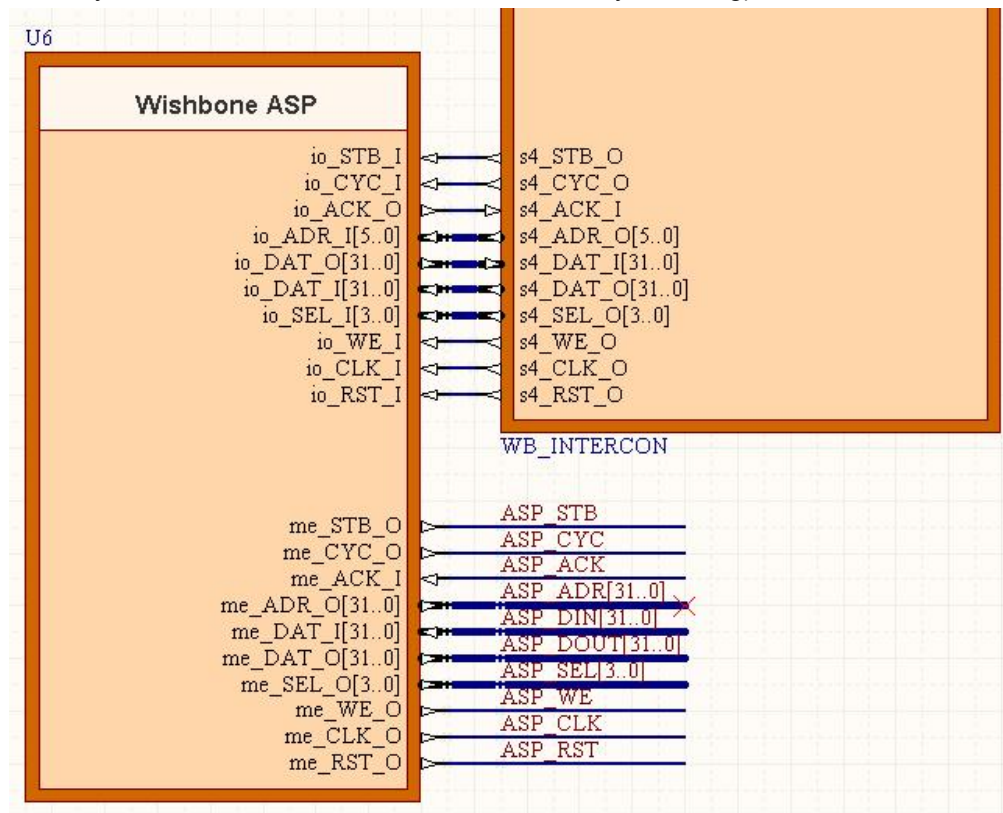
In this exercise we will add an Application Specific Processor, WB_ASP to our design. The WB_ASP peripheral is used as a 'container' for C source functions that are implemented in hardware through use of the C-to-Hardware Compiler.

Wired into an FPGA design just like any other peripheral, the WB_ASP enables a host processor access and control over hardware-compiled functions within. These functions will populate the WB_ASP once the design project has been compiled and synthesized.

When a hardware function is called, the processor simply transfers values for the function's parameters to the WB_ASP, starts the function and waits for it to return. If the hardware function delivers a return value, this will be read back by the host processor, to be used in the calling software routine.

1. From the Libraries panel, select the `FPGA_Peripherals.IntLib` file.
2. Locate the WB_ASP peripheral and select **Place WB_ASP**

3. Designate the component **U6** and place the component so the 'io_' set of signal pins align with the newly created port on the Wishbone Interconnect component **U1**.
4. Wire the component as it appears in the following figure (you may need to increase the size of your schematic document to make room for your wiring).



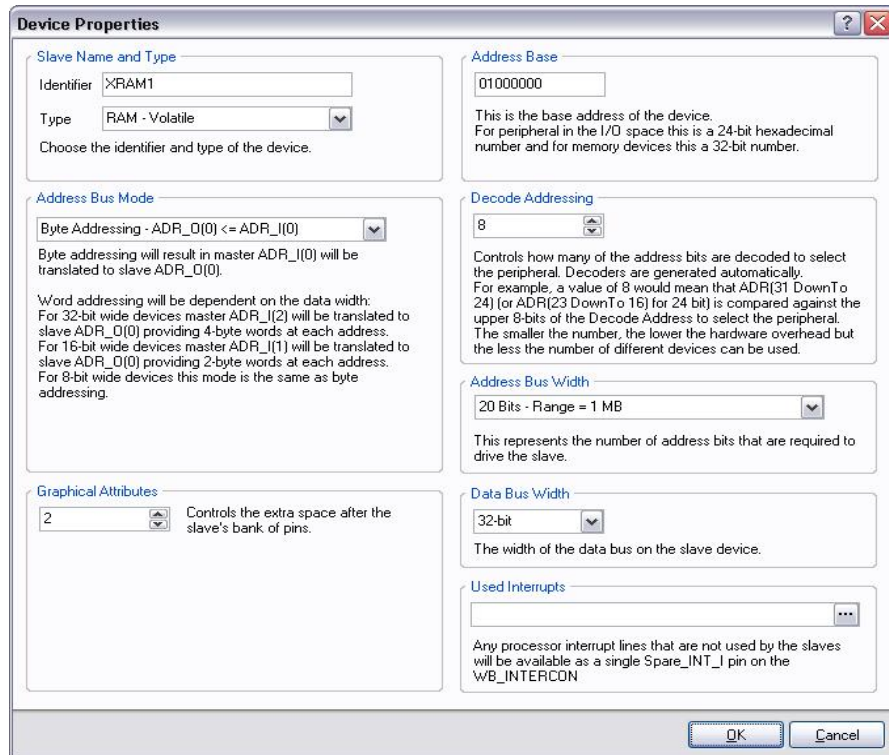
The 'io_' interface represents the interface to the processor. The 'me_' interface represents the interface to memory. Over the next few exercises we will complete the wiring, giving the ASP access to memory.

5. Update the processor peripheral memory to include the ASP by right-clicking the processor and selecting **Configure Processor Peripheral...** then selecting **Import From Schematic** to import the new configuration in the **Configure Peripherals** dialog.

12.6 Exercise 26 – Giving the ASP Access to Memory (part 1)

In this exercise we will add another Wishbone Interconnect on the memory side of our design configuring it such that we can access both memories our video capture, and video display memories.

1. From the Libraries panel, select the `FPGA_Peripherals.IntLib` file.
2. Locate the WB_INTERCON peripheral and select **Place WB_INTERCON**.
3. Designate the component **U15**
4. Open the **Component Properties** dialog and select **Configure**
5. Select **Add Device** and add a new slave interface configured as it has been in the following figure:



Device Properties

Slave Name and Type
 Identifier: XRAM1
 Type: RAM - Volatile
 Choose the identifier and type of the device.

Address Base
 01000000
 This is the base address of the device.
 For peripheral in the I/O space this is a 24-bit hexadecimal number and for memory devices this a 32-bit number.

Address Bus Mode
 Byte Addressing - $ADR\_O[0] \leq ADR\_I[0]$
 Byte addressing will result in master $ADR\_I[0]$ will be translated to slave $ADR\_O[0]$.
 Word addressing will be dependent on the data width:
 For 32-bit wide devices master $ADR\_I[2]$ will be translated to slave $ADR\_O[0]$ providing 4-byte words at each address.
 For 16-bit wide devices master $ADR\_I[1]$ will be translated to slave $ADR\_O[0]$ providing 2-byte words at each address.
 For 8-bit wide devices this mode is the same as byte addressing.

Decode Addressing
 8
 Controls how many of the address bits are decoded to select the peripheral. Decoders are generated automatically.
 For example, a value of 8 would mean that $ADR[31 \text{ DownTo } 24]$ (or $ADR[23 \text{ DownTo } 16]$ for 24 bit) is compared against the upper 8-bits of the Decode Address to select the peripheral.
 The smaller the number, the lower the hardware overhead but the less the number of different devices can be used.

Address Bus Width
 20 Bits - Range = 1 MB
 This represents the number of address bits that are required to drive the slave.

Data Bus Width
 32-bit
 The width of the data bus on the slave device.

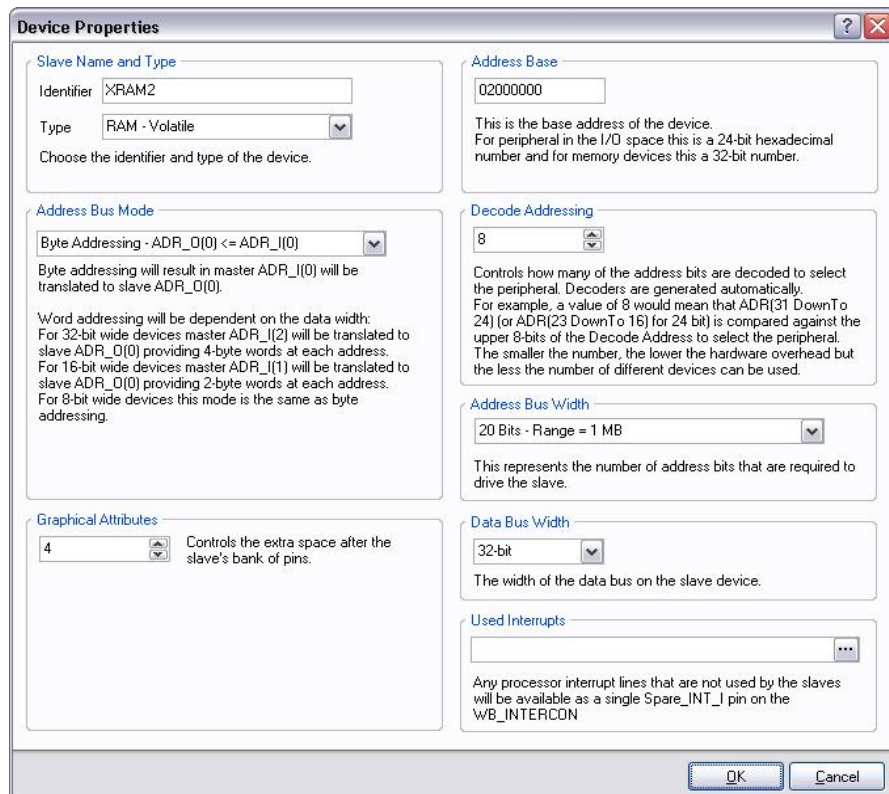
Used Interrupts
 ...
 Any processor interrupt lines that are not used by the slaves will be available as a single Spare_INT_I pin on the WB_INTERCON

Graphical Attributes
 2 Controls the extra space after the slave's bank of pins.

OK Cancel

Figure 61. Specifying the ASP interface to XRAM1

6. Click **OK** to return to the **Configure U15 (Wishbone Intercon)** dialog.
7. Select **Add Device** and add a new slave interface configured as it has been in the following figure:



Device Properties

Slave Name and Type
 Identifier: XRAM2
 Type: RAM - Volatile
 Choose the identifier and type of the device.

Address Base
 02000000
 This is the base address of the device.
 For peripheral in the I/O space this is a 24-bit hexadecimal number and for memory devices this a 32-bit number.

Address Bus Mode
 Byte Addressing - $ADR\_O[0] \leq ADR\_I[0]$
 Byte addressing will result in master $ADR\_I[0]$ will be translated to slave $ADR\_O[0]$.
 Word addressing will be dependent on the data width:
 For 32-bit wide devices master $ADR\_I[2]$ will be translated to slave $ADR\_O[0]$ providing 4-byte words at each address.
 For 16-bit wide devices master $ADR\_I[1]$ will be translated to slave $ADR\_O[0]$ providing 2-byte words at each address.
 For 8-bit wide devices this mode is the same as byte addressing.

Decode Addressing
 8
 Controls how many of the address bits are decoded to select the peripheral. Decoders are generated automatically.
 For example, a value of 8 would mean that $ADR[31 \text{ DownTo } 24]$ (or $ADR[23 \text{ DownTo } 16]$ for 24 bit) is compared against the upper 8-bits of the Decode Address to select the peripheral.
 The smaller the number, the lower the hardware overhead but the less the number of different devices can be used.

Address Bus Width
 20 Bits - Range = 1 MB
 This represents the number of address bits that are required to drive the slave.

Data Bus Width
 32-bit
 The width of the data bus on the slave device.

Used Interrupts
 ...
 Any processor interrupt lines that are not used by the slaves will be available as a single Spare_INT_I pin on the WB_INTERCON

Graphical Attributes
 4 Controls the extra space after the slave's bank of pins.

OK Cancel

Figure 62. Specifying the ASP interface to XRAM2

8. Click **OK** to return to the **Configure U15 (Wishbone Intercon)** dialog.
9. Click **OK** to close the **Configure U15 (Wishbone Intercon)** dialog.
10. Wire the design as it appears Figure 63
11. Save your work.

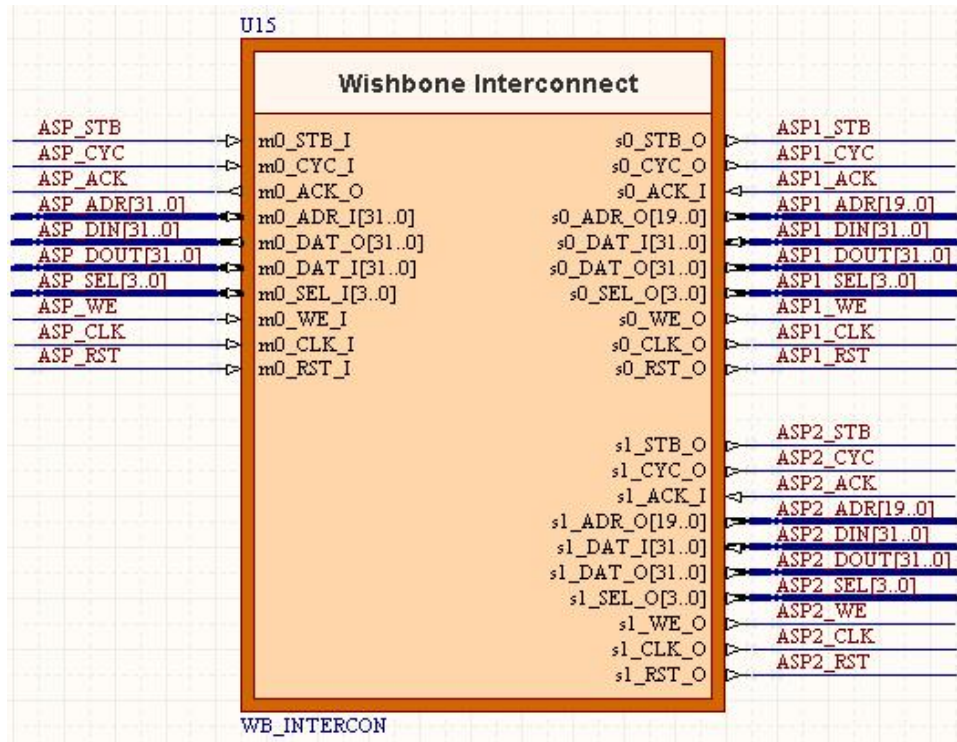


Figure 63. Wiring the ASP to interconnect to both memories

12.7 Exercise 27 – Giving the ASP Access to Memory (part 2)

In this exercise we will give each of the two Wishbone Multi Master components an interface that allows the ASP access to the display and capture memories.

1. Right Click the Wishbone Multi Master component **U3** and select **Configure U3 (WB_MULTIMASTER)**...
2. We will need to add an additional master interface to component **U3**. Configure U3 as it appears in the following dialog.

Configure U3 (Wishbone MultiMaster)

Type
☐ Round Robin ☒ Priority
 This sets the way the masters contest for the resource

Masters
 3
 This represents the number of masters on the device

Priority Order

Name	Spaces After Pins	Priority
MCU	4	2
VIDEO	5	1
ASP	4	3

The priority order of masters from highest to lowest. Use the buttons to lower and raise priority. Edit the fields to change the name of the masters and their spacings on the symbol.

Master With No Delay
 VIDEO
 When the multi-master is in idle it grants access to this master.

Address Bus Width
 20 Bits - Range = 1 MB
 This represents the number of address bits that are required to drive the slave.

Data Bus Width
 32-bit
 The width of the data bus on masters and the slave device.

Interrupts
☐ Show Interrupt Pins
 Add the interrupt pins to the wishbone interface.

Ok Cancel

Figure 64. Updating the multimaster U3

- Click OK to close the dialog.
- Right Click the Wishbone Multi Master component **U12** and select **Configure U12 (WB_MULTIMASTER)...**
- We will need to add an additional master interface to component **U12**. Configure U12 as it appears in the following dialog.

Configure U12 (Wishbone MultiMaster) ? X

Type
☐ Round Robin ☒ Priority
 This sets the way the masters contest for the resource.

Masters
 3 ▼
 This represents the number of masters on the device.

Priority Order

Name	Spaces After Pins	Priority
MCU	4	2
TFT	5	1
ASP	4	3

The priority order of masters from highest to lowest. Use the buttons to lower and raise priority. Edit the fields to change the name of the masters and their spacings on the symbol.

Master With No Delay
 TFT ▼
 When the multi-master is in idle it grants access to this master.

Address Bus Width
 20 Bits - Range = 1 MB ▼
 This represents the number of address bits that are required to drive the slave.

Data Bus Width
 32-bit ▼
 The width of the data bus on masters and the slave device.

Interrupts
☐ Show Interrupt Pins
 Add the interrupt pins to the wishbone interface.

Ok Cancel

Figure 65. Updating the multimaster U12

6. Click OK to close the dialog.
7. Wire the two components as they appear in Figure 66 and Figure 67.
8. Save your work.

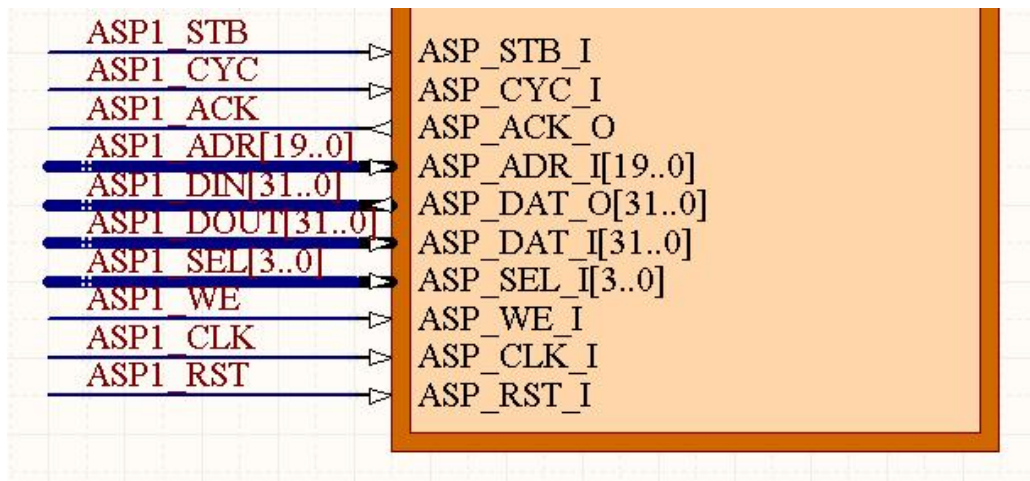


Figure 66. Updated wiring for wishbone multimaster U3

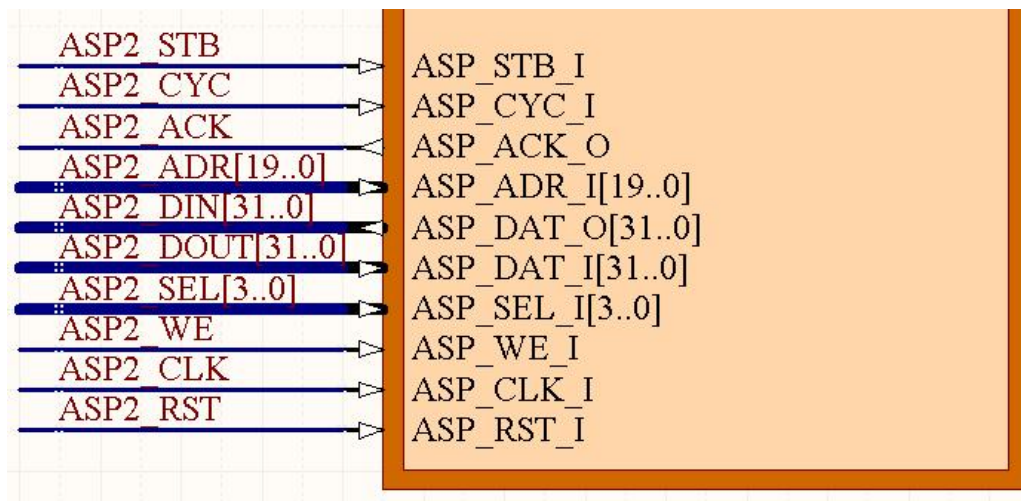


Figure 67. Updated wiring for wishbone multimaster U3

12.8 Configuring the ASP

To begin realizing the benefits of C to Hardware we must first configure our Wishbone ASP component. To configure, right click the component in schematic and select **Configure U6 (WB_ASP)**. This will launch the **Configure U6 (WB_ASP Properties)** dialog seen in Figure 68.

Configure U6 (WB_ASP Properties)

ASP Options

Processor: U2

Use the following options to enable or disable acceleration. You can do this at either the FPGA or the processor code level.

The Generate ASP option controls whether any FPGA logic is generated. Disable this option to globally disable the ASP and not generate any FPGA logic.

The Use ASP from Software option controls whether the code running on the processor will call the ASP or not. Disable this option to disable the use of the ASP while retaining the ASP logic in the FPGA.

☒ Generate ASP

☐ Use ASP from Software

Parameter Bus Options

Select the size of the address bus width. It should be large enough to address the parameters of the largest function in the design.

Address Bus Width: 6

Select the number of spaces after the parameter bus on the schematic.

Extra space: 4

Symbols In Hardware

Global Variable	Allocate in Hardware
bt656_settings	☐
costab	☐
imgbuf	☐
Ports	☐
sintab	☐
tft	☐

Function	Implement in Hardw...	Export to Software
init	☐	☐
main	☐	☐
rotate	☐	☐
set_tabs	☐	☐

OK Cancel

Figure 68. Configuring the wishbone ASP

The following table describes the various options available in the configuration dialog.

Option	Description
Processor	Use this field to specify the processor, by designator, that is connected to the WB_ASP and which can therefore call functions that have been generated in hardware.

Option	Description
Generate ASP	This option provides the ability to enable or disable generation of hardware-compiled functions. With this option enabled, the C-to-Hardware Compiler will be invoked when you compile and synthesize your design project. All functions that have been enabled for implementation in hardware will be created as electronic circuits in the FPGA fabric.
Use ASP from Software	This option enables you to control, on a global level, whether functions compiled into hardware will be called by software-based functions running within the processor. If this option is disabled, the embedded compiler will generate the functions in software and these will be used.
Address Bus Width	Use this field to specify the width of the WB_ASP's host interface address bus (io_ADR_I). This bus is used by the processor to access and pass values to parameters in a hardware function, and also to access and read back a function's return value, where applicable. The width specified here should be large enough to include all parameters of the largest function being implemented in hardware.
Extra Space	Use this field to specify the amount of blank space between the host processor and external memory interfaces to the device.
Symbols in Hardware – Global Variables	This list reflects all global variables present in the linked embedded software project. If you want to have a variable allocated in hardware, simply enable the corresponding check box in the Allocate in Hardware column. Such a variable will be allocated in ASP block RAM by the CHC Compiler. Access to this memory is much faster, in comparison to storage allocation in block RAM outside of the ASP by the Embedded Compiler. The list supports standard multi-select features, allowing you to quickly select multiple variables. Once selected, use the available right-click context menu commands to quickly enable (Push to Hardware) or disable (Remove from Hardware) the corresponding Allocate in Hardware option. Note that a global variable that is allocated in hardware <i>can only be accessed by a function that has also been implemented in hardware</i>. Such a variable cannot be called from a software-based function running on the host processor.

Option	Description
Symbols In Hardware – Function	This list reflects all functions present in the linked embedded software project. If you want to implement a function in hardware - generated by the CHC Compiler as part of the ASP - simply enable the corresponding check box in the Implement in Hardware column. Should you wish to be able to call that hardware function from within the software running on the host processor, ensure that the corresponding check box in the Export to Software column is also enabled. The list supports standard multi-select features, allowing you to quickly select multiple functions. Once selected, use the available right-click context menu commands to: • Push to Hardware - enable Implement in Hardware option for each function in the selection • Remove from Hardware - disable Implement in Hardware option for each function in the selection • Push and Export to Hardware - enable Implement in Hardware and Export to Software options for each function in the selection • Unexport from Hardware - disable Export to Software option for each function in the selection.

12.9 Exercise 28 – Accelerating our performance with ASP

In this exercise we will configure the ASP to 'offload' some of the functionality of our design (previously done in software) into the FPGA hardware.

1. Right click the WB_ASP component U6 in the schematic and select Configure U6 (WB_ASP)... The following table describes the various options found in this dialog.
2. Configure the WB_ASP **U6** to match the parameters in figure 69.

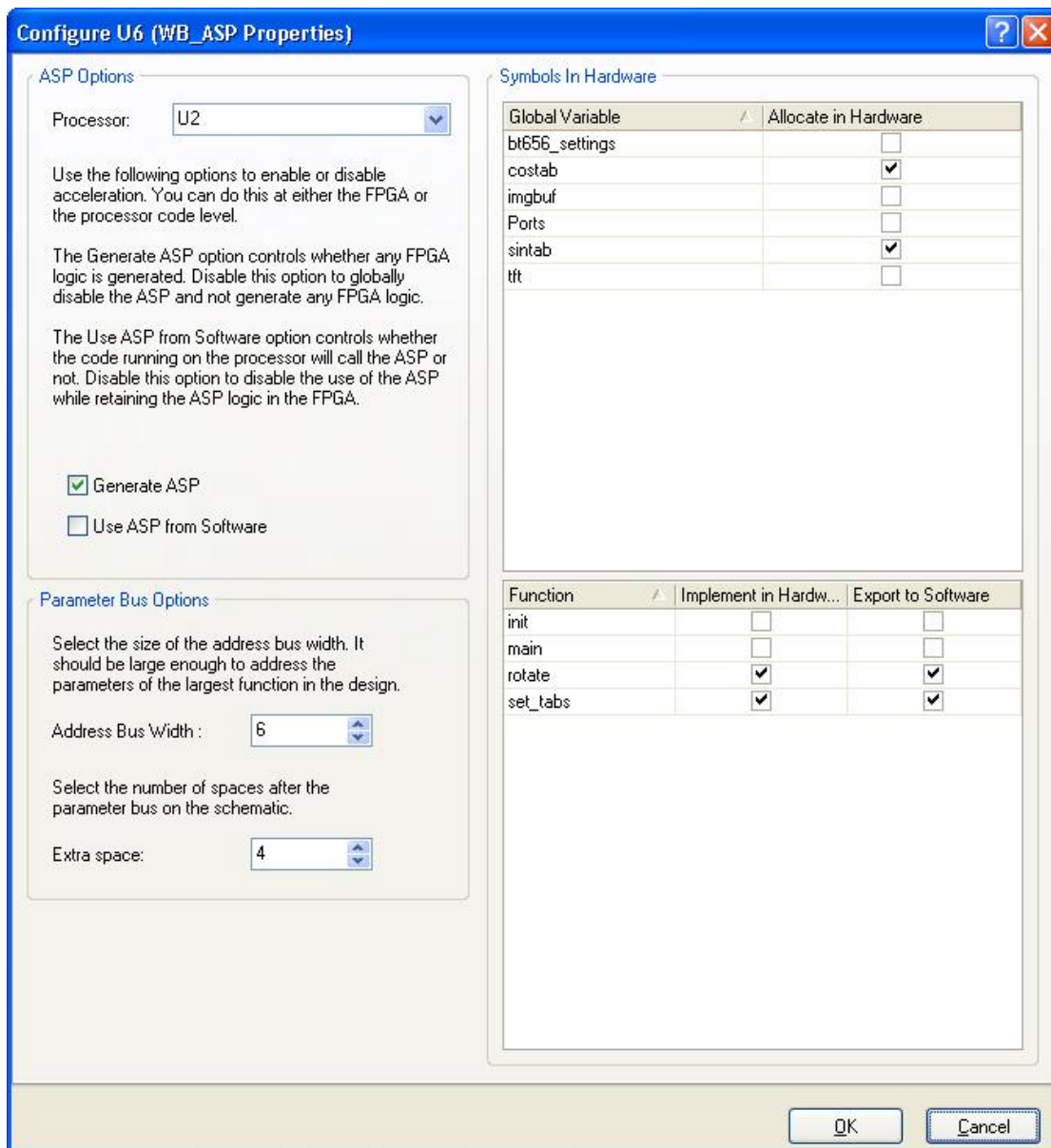


Figure 69. Configuring the wishbone ASP

What we have done is move the two functions, `rotate` and `set_tabs` into hardware. The `rotate` function performs the rotation of the video, and the `set_tabs` function builds our Sine and Cosine tables. The global variables `costab` and `sintab` have also been moved to hardware as these are used by both of these functions and can be accessed more quickly when accessed from ASP block RAM.

- Click OK to close the dialog.
- Make sure your Desktop NanoBoard is connected to your PC and powered on.
- Select **View >> Devices View** or click on the **Devices View** icon  in the toolbar.
- Ensure that the **Live** checkbox is checked. You should see a picture of the Desktop NanoBoard in the upper region of the display and an icon of the Spartan3 FPGA in the middle region.
- In the drop down list just below the Spartan3 icon, ensure that the **CHC_image_rotation / NBD2DSK01_07_DB30_04** project / configuration pair is selected.

8. Locate the **Compile, Synthesize, Build, Program FPGA** buttons running left to right just below the Desktop NanoBoard icon. Click once on the words **Program FPGA** to begin the build process.
9. Once the design has completed the build process, return to the **Devices** view and right click the **Digital IO** instrument and select **Instrument**.
10. From the **Instrument Rack**, adjust the **Rot[7..0]** and **Zoom[7..0]** sliders to see the performance improvement when utilizing C to Hardware.

13 OpenBus

Until now, our discussions of FPGA design has really centered around schematic, with all devices in the system laid out on a schematic sheet or captured in a hardware description language such as VHDL or Verilog. Such designs suffer an inherent complexity, in terms of readability and as importantly, from a wiring / connectivity and configuration perspective.

Using Altium Designer's OpenBus System we aim to greatly reduce such complexity. The fundamental purpose of this system is to represent the processor-peripheral interconnections in a much more abstract way. It achieves this by providing an environment in which to create your system that is highly intuitive, streamlined, and less prone to error.

OpenBus System is created and managed using Altium Designer's OpenBus Editor. The OpenBus Editor has the familiar look and feel of Altium Designer's Schematic Editor, with its own unique set of resources for creation of an OpenBus System. Filtering and inspection are provided courtesy of the **OpenBus Filter**, **OpenBus Inspector** and **OpenBus List** panels, accessed from the **OpenBus** panel access button, to the bottom-right of the main design window. These resources provide a raised abstraction level for creating your processor-based design in a more visual and intuitive way.

13.1 Creating an OpenBus version of our design

Creating an OpenBus design begins like any other FPGA development effort in Altium Designer, with the creation of a new FPGA Project. To that project, source documents are added and may include any combination of Schematics, VHDL or Verilog files, as well as OpenBus documents.

Any FPGA design project that uses this system must have a top-level schematic as all interface circuitry remains on the top-sheet. The main processor system is defined in a separate, OpenBus System document (*.OpenBus) and connectivity between the two is made through a sheet symbol placed on the schematic.

In this section we will create our equivalent design using OpenBus, though we will, for simplicity's sake, eliminate the virtual instruments and instead simply focus on getting the image onto the TFT.

13.2 Exercise 29 – Beginning an FPGA Project Using OpenBus

1. Select **File»New»Project»FPGA Project** to create a new, blank FPGA project
2. Right click the project `FPGA project FPGA_Project1.PrjFpg` in the **Projects** panel and select **Add New to Project»OpenBus System Document**
3. Right click the project `FPGA project FPGA_Project1.PrjFpg` in the **Projects** panel and select **Add New to Project»Schematic**
4. Right click the project `FPGA project FPGA_Project1.PrjFpg` in the **Projects** panel and select **Save Project As...** This will prompt you to first save the schematic, then the OpenBus document, and lastly the FPGA project document. Save these as `Video_Capture_Schematic.SchDoc`, `Video_Capture_OpenBus.OpenBus`, and `Video_Capture.PrjFpg` respectively.
5. From the **Projects** panel, select the OpenBus document to give it focus in the main workspace.

13.3 OpenBus Devices

The starting point for any OpenBus System document is the placement of the required devices that will constitute the system. These OpenBus components, as they are called, are placed from the **OpenBus Palette** panel. Open this panel from the **OpenBus** button to the bottom-right of the main design window.

The panel contains graphical representations of devices available for FPGA design in Altium Designer, grouped by function:

- Connectors
- Processors
- Processor Wrappers
- Memories
- Peripherals

Placement of OpenBus components is simply a case of clicking the required entry in the **OpenBus Palette** panel and then placing at the desired location in the workspace. Familiar schematic placement controls, such as flipping and rotating allow for fine tuning as needed.

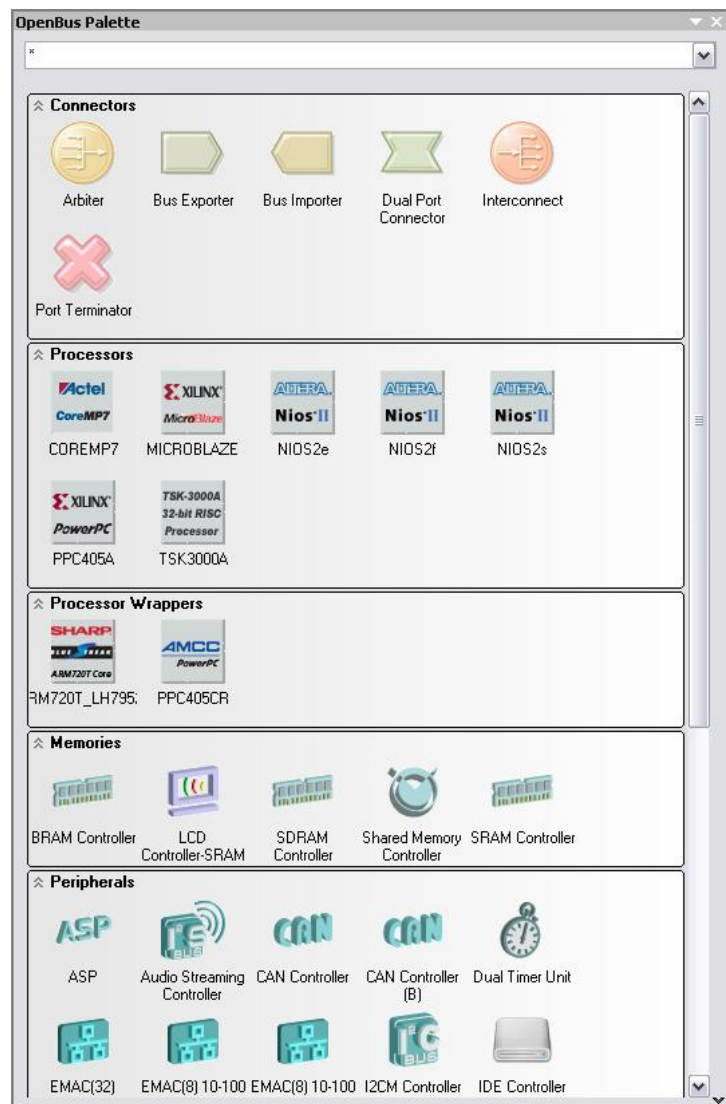


Figure 70. The OpenBus palette.

13.4 Exercise 30 – Placing Open Bus Components (part 1)

Recall that in our original schematic we used only a handful of cores to create the basic functionality required by our design. This included:

- BT656 Controller
- I²C Controller
- Wishbone Port IO
- Wishbone Display Driver
- TSK3000A
- 2 x SRAM Controllers

We then also used:

- 2 x Configurable Wishbone Interconnect components
- 2 x Wishbone MultiMaster components

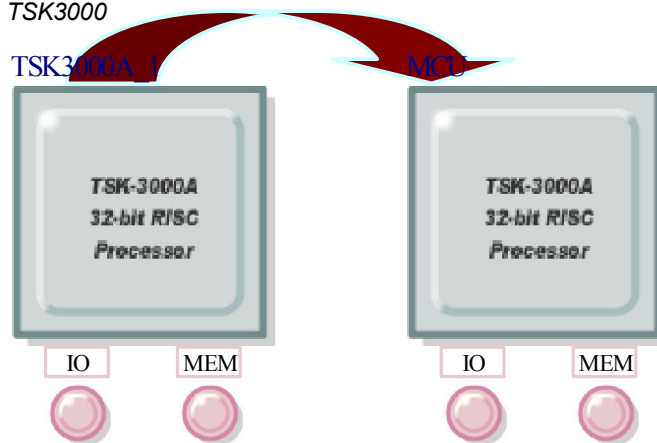
...To manage the basic interconnectivity in our design.

Lastly we used a handful of Virtual Instruments that completed the user interface to our design.

In this exercise we will begin to place the basic components of our OpenBus system.

1. Ensure the OpenBus document `VideoCapture.OpenBus` is the currently focused document in the workspace.
2. Locate the **OpenBus** panel either using the **OpenBus** button on the lower right of the workspace or by selecting **View»Workspace Panels»OpenBus»OpenBus Palette**.
3. Locate the **TSK3000A** in the **Processors** section of the **OpenBus Palette** and single-click it to switch to placement mode.
4. Move the cursor over the center of the OpenBus document and click once to place the component into the document.
5. Single-click the component text **TSK3000A_1** and hit **F2** to perform an In-Place edit of the text. Change this to **MCU** as seen in the Figure 71.

Figure 71. Naming the TSK3000



6. The two red dots on the outside of the component body represent the two master interfaces to the host processor. These can be moved with respect to the component body by simply clicking and dragging.



Reposition the IO and Memory Interfaces as they appear in the following figure. These will automatically snap into position as required.

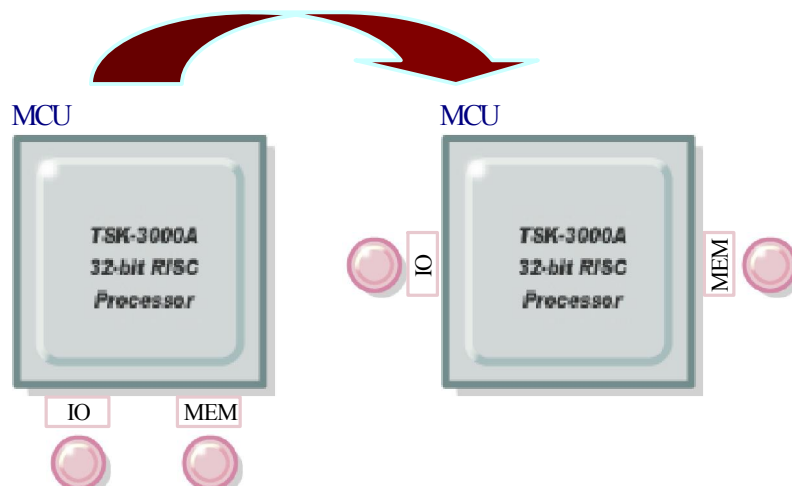


Figure 72. Repositioning the OpenBus ports on the MCU

7. Return to the **OpenBus** panel and locate the **Video Capture Controller** component in the **Peripherals** section. Select this component and place it into the OpenBus document.
8. Single-click the component text **BT656_1** and hit **F2** to perform an In-Place edit of the text. Change this to **VIDEO** as seen in Figure 73.

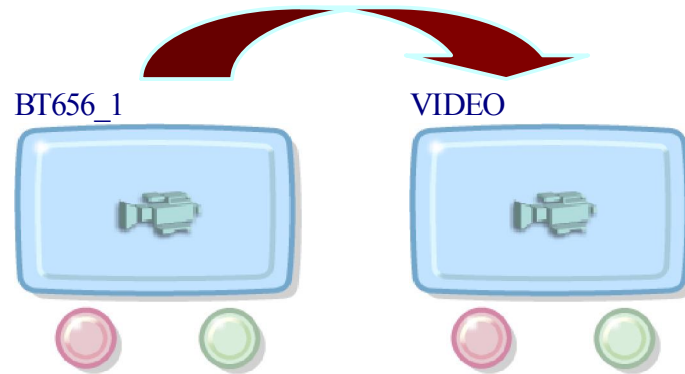


Figure 73. Renaming the video capture controller

9. The Red OpenBus port represents a Wishbone Master interface. This interface is used to connect the Video Capture controller to memory.

The Green OpenBus port represents a Wishbone Slave interface. This interface is used to connect the device to the host processor, via the Wishbone Interconnect component.

Reposition the Green and Red OpenBus ports as seen in the following figure.

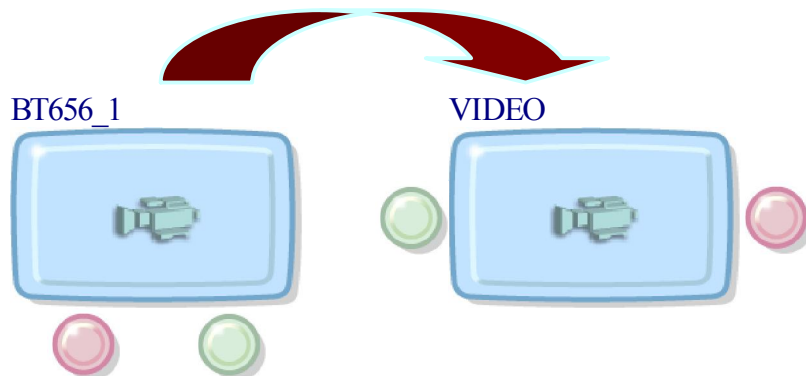


Figure 74. Repositioning OpenBus ports on the capture controller

10. Return to the **OpenBus** panel and locate the **ASP** component in the **Peripherals** section. Select this component and place it into the OpenBus document.
11. Single-click the component text **WB_ASP** and hit **F2** to perform an In-Place edit of the text. Change this to **ASP**.
12. Reposition the Green and Red OpenBus ports consistent with the Video Capture Controller.
13. Return to the **OpenBus** panel and locate the **VGA 32-Bit TFT Controller** component in the **Peripherals** section. Select this component and place it into the OpenBus document.
14. Single-click the component text **VGA32_TFT_1** and hit **F2** to perform an In-Place edit of the text. Change this to **TFT**.
- 15.

16. Reposition the Green and Red OpenBus ports consistent with the Video Capture Controller.
17. Return to the **OpenBus** panel and locate the **I²C** component in the **Peripherals** section. Select this component and place it into the OpenBus document.
18. Single-click the component text **I2CM_W_1** and hit **F2** to perform an In-Place edit of the text. Change this to **I2C**.
19. Position the Green OpenBus port to the right of the component as it appears in the following figure:

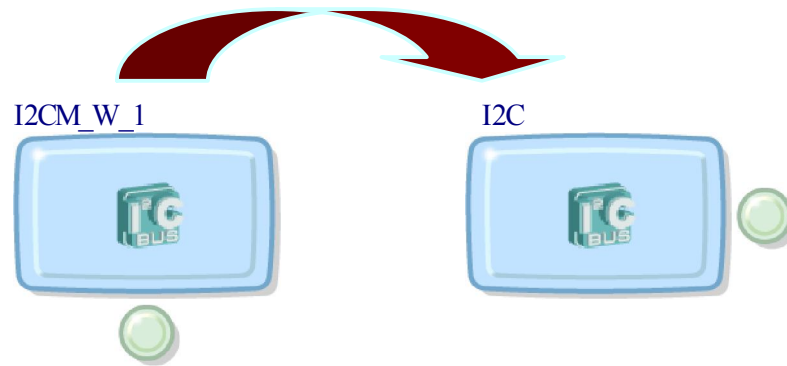


Figure 75. Renaming the I2C controller

20. Return to the **OpenBus** panel and locate the **Port IO** component in the **Peripherals** section. Select this component and place it into the OpenBus document.
21. Single-click the component text **WB_PRTIO_2** and hit **F2** to perform an In-Place edit of the text. Change this to **GPIO**.
22. Finally, reposition the OpenBus components in the schematic as they appear in the following **Figure**.

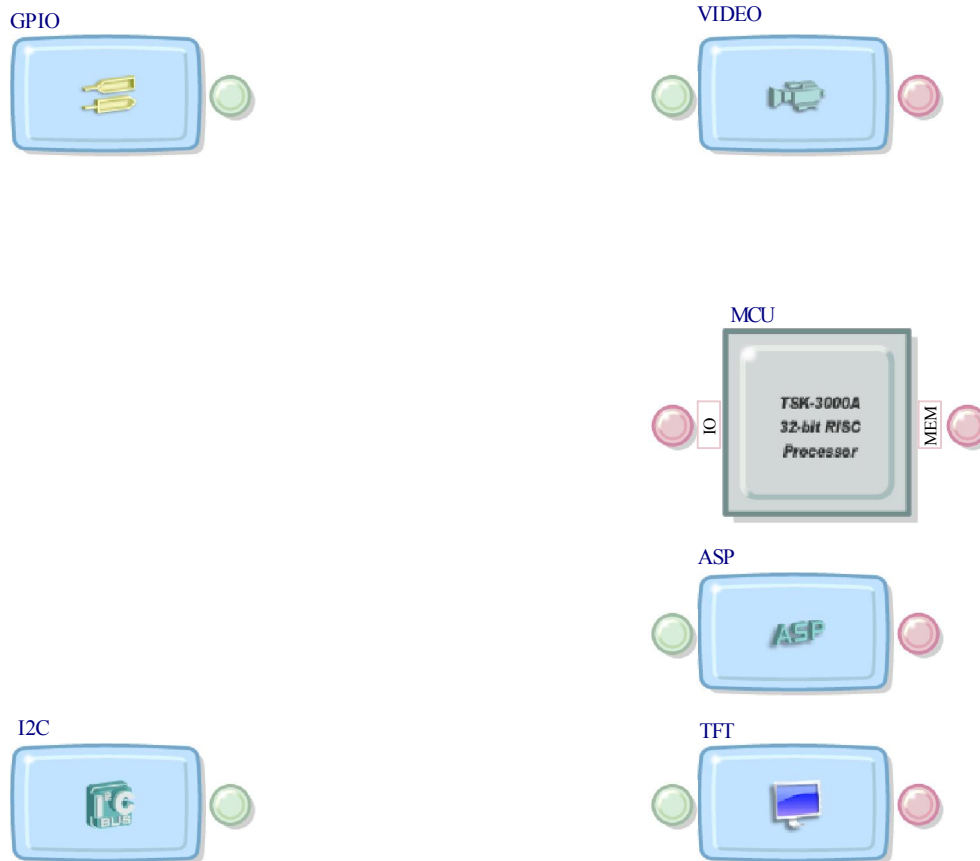


Figure 76. Positioning the processor and peripherals

The alignment tools used in a traditional Altium Designer schematic are also available under OpenBus. These are all accessible under the single key shortcut 'A'. These tools make the process of placing and aligning the OpenBus components faster and easier.

13.5 Exercise 31 – Using an Interconnect

As in schematic, OpenBus designs utilize Interconnect components to interconnect any number of slave devices to a single master. In this exercise we will place our Interconnect component and connect our peripheral devices to the host processor. You will find this process to be greatly simplified under the OpenBus paradigm.

1. From the **Connectors** section of the **OpenBus Palette**, select the **Interconnect** component and place the component such that the green Slave interface aligns directly to the left of the host processor's red Master interface labeled **IO**.
2. Single-click the component text **WB_INTERCON_1** and hit **F2** to perform an In-Place edit of the text. Change this to **I_IO**.
3. Locate the OpenBus toolbar along the top of the main workspace. The following table describes each of the items in this toolbar.



Toolbar Item	Command	Description
	Link OpenBus Ports	Use this command to link the ports of devices within your OpenBus System. When you enter link addition mode, all currently unlinked ports will be filtered, with all other elements in the system dimmed. If you click to start a link on a master port, all currently unlinked slave ports will be filtered and available for choosing the termination point of the link. Conversely, if you click to start a link from a slave port, all currently unlinked master ports will be filtered and made available. To remove a link, simply click to select it in the workspace, then press the Delete key.
	Add OpenBus Ports	Use this command to add one or more additional ports to an Interconnect or Arbiter component in the OpenBus System document. After launching the command, a dimmed port shape will appear floating on the cursor. As you move the shape next to the perimeter of a component, it will become solid - darker gray with a blue outline. Position the new port as required and click to effect placement. When adding the port to an Interconnect component, a new master port will be added. When adding the port to an Arbiter component, a new slave port will be added. Continue adding further ports as required, or right-click or press Esc to exit port addition mode.
	Normalize OpenBus Link(s) Shape	This command allows you to quickly normalize the shape of one or more links in the OpenBus System document. Ensure that all links to be normalized are selected prior to launching the command. Once the command is launched, all links in the selection will revert back to the standard shapes they had when first placed.

Toolbar Item	Command	Description
	Flip OpenBus Link(s) Shape	This command allows you to quickly flip one or more links in the OpenBus System document. Ensure that all links to be flipped are selected prior to launching the command.
	Straighten OpenBus Link(s) Shape	Allows you to quickly straighten one or more links in the OpenBus System document. Ensure that all links to be straightened are selected prior to launching the command. Once the command is launched, all links in the selection will be straightened, going from source master port to destination slave port.
	Manage OpenBus Signals and Interrupts	Use this command to access the OpenBus Signal Manager dialog. This dialog is used to manage the clock, reset and interrupt lines for the system. The export of interrupt lines outside of the OpenBus System document is handled using this dialog. The dialog provides an External connection summary tab. This lists each of the peripheral or memory devices used in the system, and the external interface signals associated with them. These signals are the ports that correspond to the sheet entries of the sheet symbol on the top-level schematic sheet - the method by which the OpenBus System is hooked into the overall FPGA design.

- For our design we will need to add additional Master ports to our Interconnect component. These will enable us to connect the additional peripherals to our host processor.

Press the **Add OpenBus Port** button  in the **OpenBus** toolbar and click once over the red Master port on the **Interconnect** component. Notice how the ports are automatically aligned.

- Place 3 additional OpenBus ports to bring the total to 1 Slave Port (the interface to the host processor) and 5 Master Ports (the interfaces to our various peripherals).
- Simply clicking and dragging on any port will allow you to reposition it around the outside of the body of the Interconnect device. Align the ports to match the following figure.

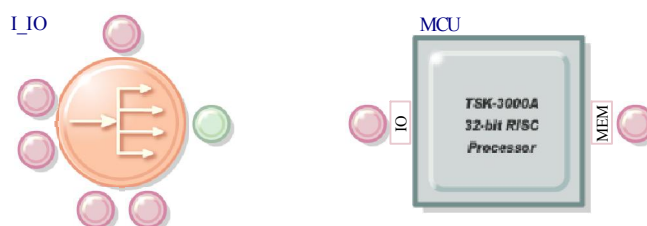


Figure 77. Positioning the ports on the interconnect component

- Save your work

13.6 Exercise 32 – Linking OpenBus Ports

In this exercise we will be linking the OpenBus ports to one another on the peripheral half of our design. This can best be likened to wiring the Wishbone Interfaces of schematic components to one another in the schematic editor; however this process has been even further simplified under OpenBus.

1. From the OpenBus toolbar, select the Link OpenBus Ports command  and the cursor will change to a cross-hair.
2. Click on one of the Master ports on the Interconnect component **I_IO**. Notice how only the valid Slave interfaces are available for selection. Clicking on a Slave interface will connect the two devices to one another and the signals required to connect these two components will be managed automatically by the software.

This is a key component of the OpenBus system and something not to be taken lightly.

And though what is ‘under the hood’ is always available for us to inspect, there’s little need when using the cores supplied with Altium Designer.

3. Connect each of the peripherals to the interconnect component as displayed in the figure below. Experiment using the **OpenBus** toolbar options to Normalize , Flip , and Straighten  the OpenBus connections.

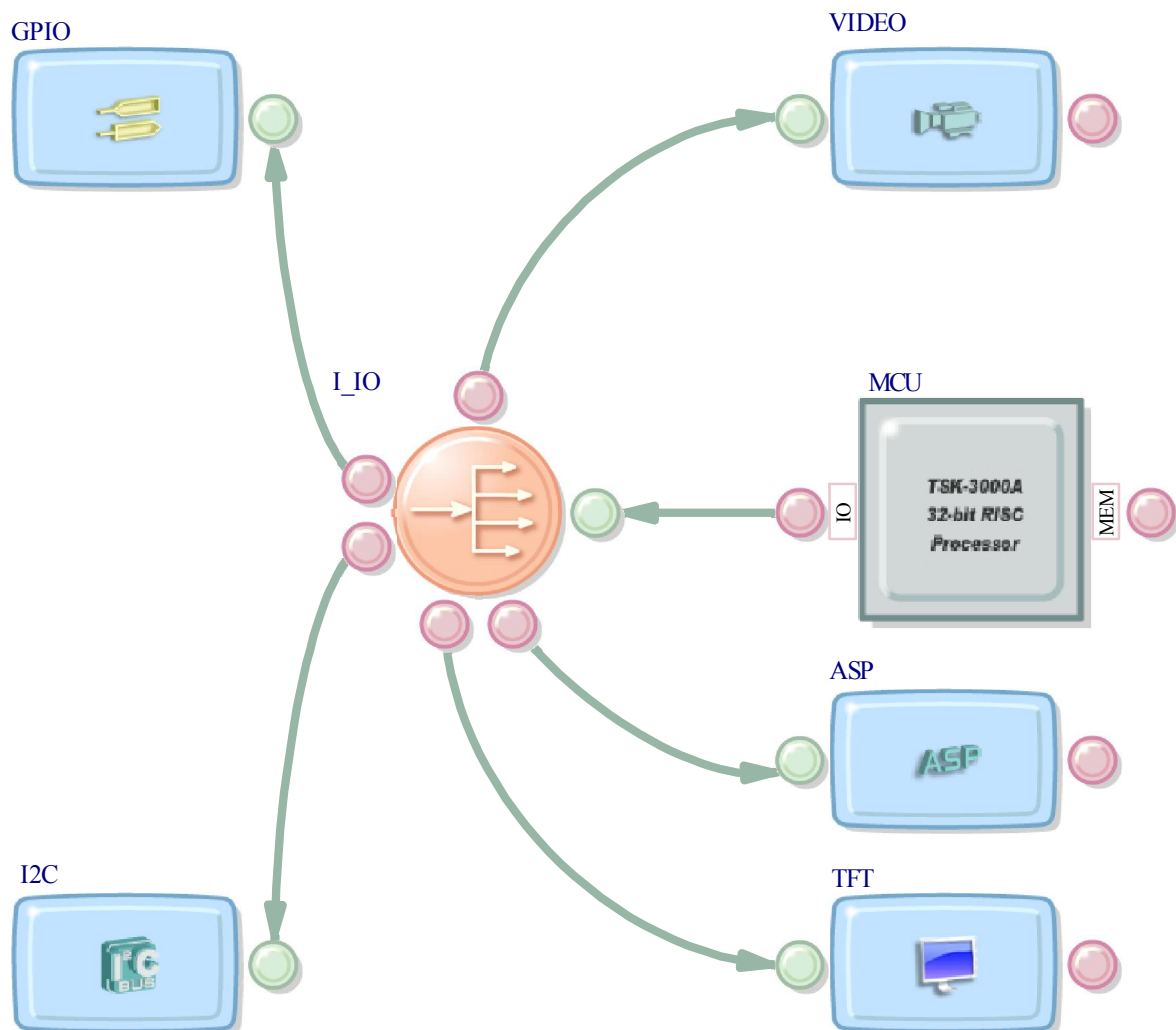


Figure 78. Connecting the peripherals

13.7 Exercise 33 – Configuring our processor under open bus

The process of configuring the host processor under OpenBus is in many ways identical to the way in which a processor is configured in schematic. In this exercise we will configure the host processor for use in our design.

1. Right click the host processor in the OpenBus document and select **Configure MCU (TSK3000A)...**
2. Configure the processor options found in the **Configure (32-bit Processors)** dialog as they appear in Figure 79.



Figure 79. Configuring the processor

3. Click OK to close the dialog.
4. Save your work.

13.8 Exercise 34 – Configuring the GPIO Component

In this exercise we will configure our GPIO component to include the ports required to connect our LEDs and Virtual Instruments to this design.

1. Right click the **GPIO** component and select **Configure GPIO (Port IO)...**
2. Configure the settings in the **Configure OpenBus Port I/O** as they appear in Figure 80.

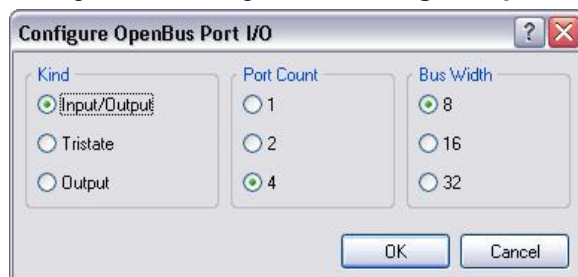


Figure 80. Configuring the Port IO

Each of the ports specified in the dialog will be converted to a sheet entry later in this design. When specifying **Input/Output** as the **Kind**, two separate sheet entries will be created, one for the input, another for the output.

3. Click OK to close the dialog.
4. Save your work.

13.9 Exercise 35 – Finalizing the Interconnect Component

In this exercise we will configure the Interconnect component, defining the characteristics of each of the peripherals as they exist in the processor's memory space. This process has been streamlined under OpenBus as many of these attributes can be inferred from the peripherals.

For example, we can infer from our Port IO that were we to select 4 ports in the GPIO configuration dialog, we would require 2 bits to address each of the 4 ports. Under OpenBus, these values are populated automatically and only require us to specify the number of bits to decode and the base address of each of the peripherals.

1. Right click the **I_IO** component and select **Configure I_IO (Interconnect)...**
2. Clicking on a field and hitting **F2** will allow you to make edits to the contents of each field. Configure the settings in the **Configure OpenBus Interconnect** dialog to match the figure below.

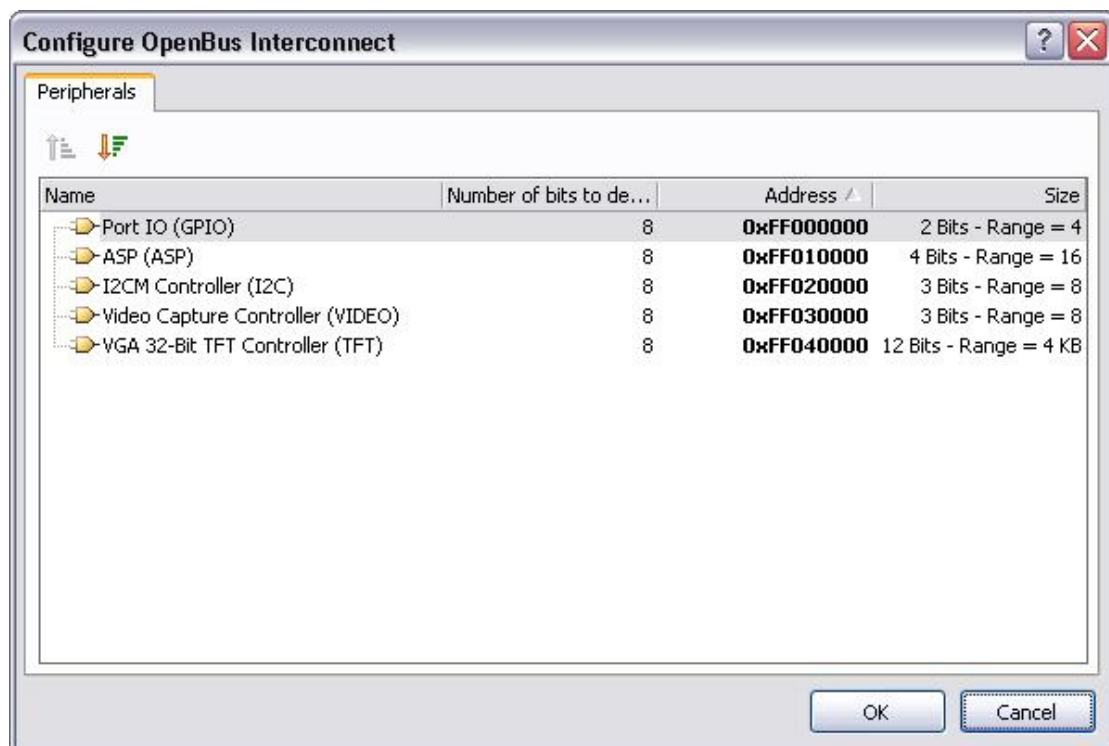


Figure 81. Configuring the OpenBus Interconnect

3. Click OK to close the dialog.
4. Save your work.

13.10 Defining the Memory Side of our System

As we've seen in the schematic portion of the course, this design requires external memory. In this section we will add the IP necessary to access memory resources on the Desktop NanoBoard.

13.11 Exercise 36 – Adding the Remaining Components

In this exercise we will add the final few parts for the OpenBus portion of our design. Below is a list of these parts, all of which should seem familiar given the earlier schematic-based approach we've already completed.

- 2 x Interconnect components (named I_MCU, I_ASP)
- 2 x Arbiter components (named MULTIMASTER_1, MULTIMASTER_2)
- 2 x SRAM Controller (named XRAM1, XRAM2)

From the **OpenBus Palette**, place the above named components as they appear in the following figure. Note the additional Master and Slave ports on the Interconnect and Arbiter components respectively.

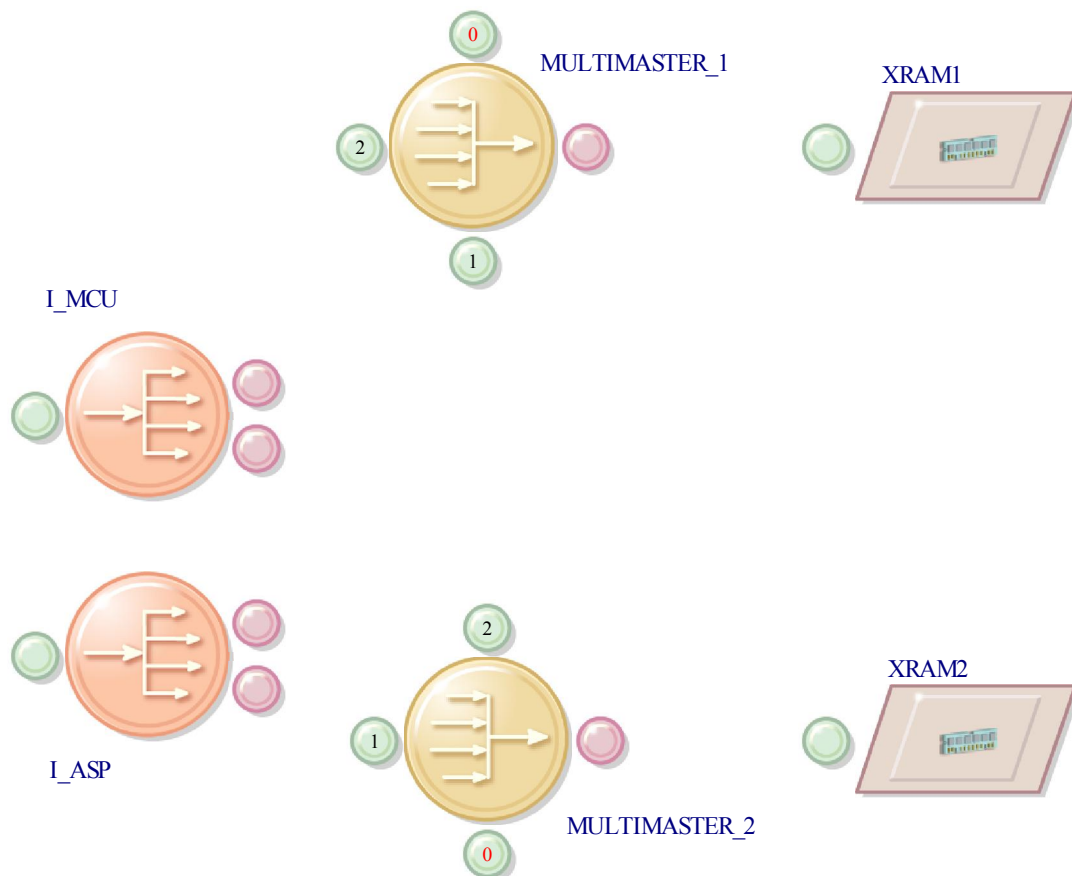


Figure 82. Placing the memory and memory interconnects / arbiters

13.12 Exercise 37 – Completing the Connectivity

In this exercise we will complete the connectivity required by the memory portion of our design. Using the toolbar command **Link OpenBus Ports**, connect the ports as they appear in the following figure.

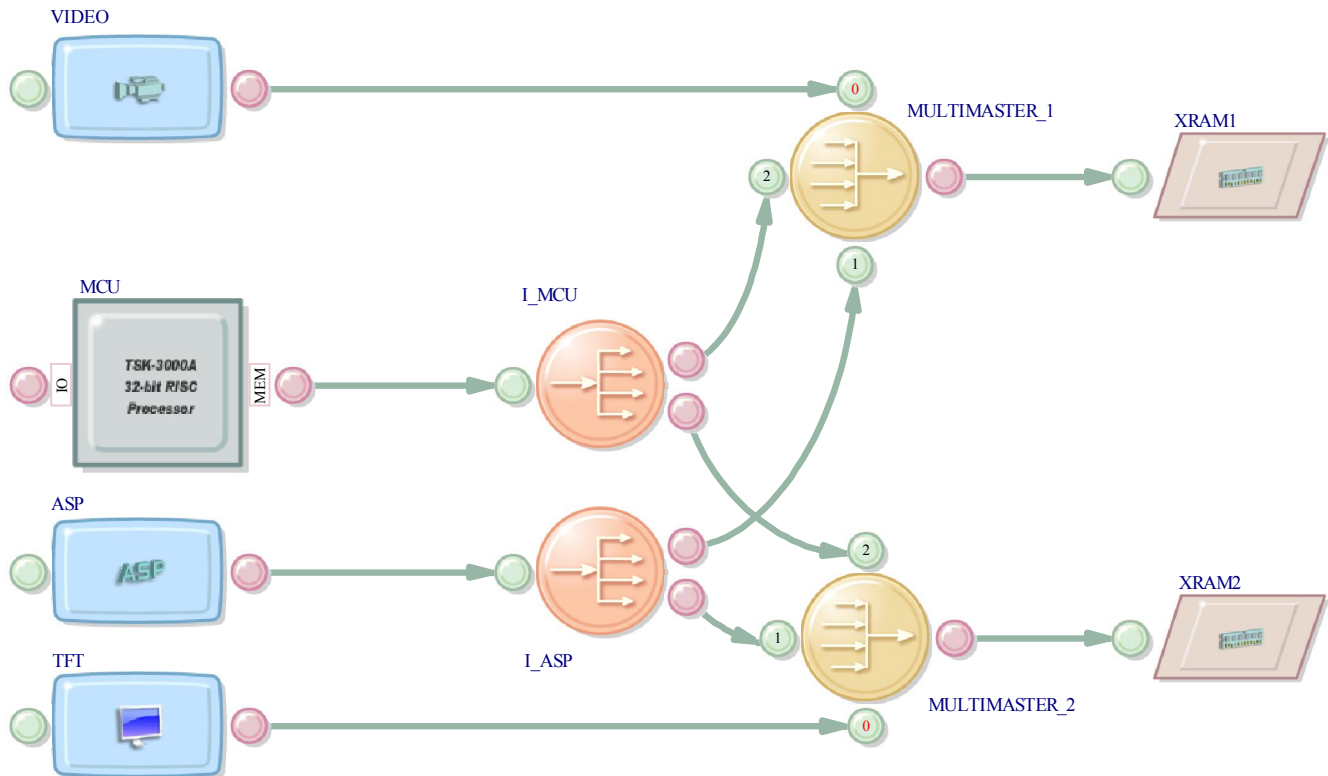


Figure 83. Connecting the memory side of the design

13.13 Exercise 38 – Configuring the SRAM Controllers

In this exercise we will configure the two SRAM Controllers for use by our design. If you recall from the schematic portion of the course, both of these controllers interfaced to SRAM on the Desktop NanoBoard. The sizes of these memories had been constrained by the devices on the board, and the requirements of the design.

1. Right click XRAM1 in the OpenBus document and select **Configure XRAM1 (SRAM Controller)** to launch the component's configuration dialog.
2. This dialog matches exactly the dialog we had used in schematic. In the **Configure (Memory Controller)** dialog, specify the settings to match the following figure.

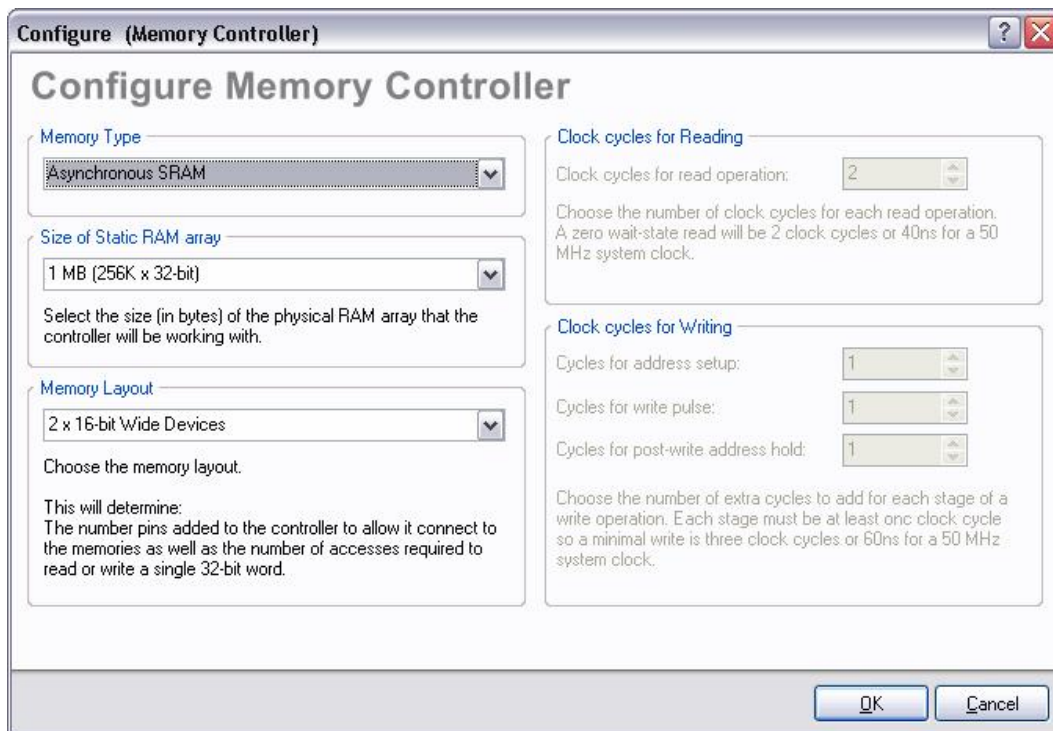


Figure 84. Configuring the Memory Controller XRAM1

3. Click OK to close the dialog and return to the OpenBus document.
4. Right click **XRAM2** in the OpenBus document and select **Configure XRAM2 (SRAM Controller)** to launch the component's configuration dialog.
5. In the **Configure (Memory Controller)** dialog, specify the settings to match the following figure.

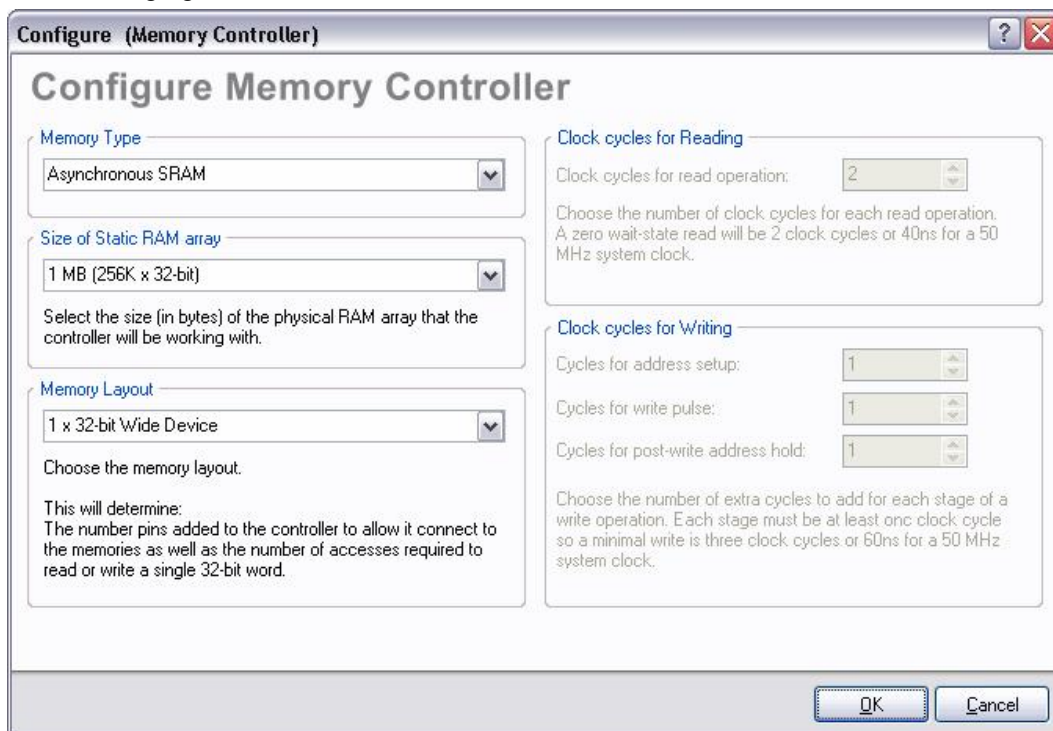


Figure 85. Configuring the Memory Controller XRAM2

6. Click OK to close the dialog and return to the OpenBus document.
7. Save your work.

13.14 Exercise 39 – Configuring the Arbiters

In this exercise we will configure the two **Arbiter** components for use by our design. If you recall from the schematic portion of the course, there was one arbiter for each of the two memories. These manage the processor, ASP, and capture and display cores access to memory.

Under the OpenBus system, the interface to the arbiters has been simplified considerably. Because of the addition of the OpenBus ports, we are no longer required to add additional items to the arbiter and manage naming and pin spacing. Instead, we are only required to select the way masters contest for the slave resource and specify the master with no delay.

1. Right click **MULTIMASTER_1** in the OpenBus document and select **Configure MULTIMASTER_1 (Arbiter)** to launch the component's configuration dialog.
2. In the **Configure OpenBus Arbiter** dialog, specify the settings to match the following figure.



Figure 86. Configuring the OpenBus arbiter

3. Click OK to close the dialog and return to the OpenBus document.
4. Right click **MULTIMASTER_2** in the OpenBus document and select **Configure MULTIMASTER_2 (Arbiter)** to launch the component's configuration dialog.
5. In the **Configure OpenBus Arbiter** dialog, specify the settings to match the following figure.



Figure 87. Configuring the OpenBus arbiter

13.15 Exercise 40 – Configuring the Interconnects

In this exercise we will configure the two **Interconnect** components for use by our design. If you recall from the schematic portion of the course, there were two interconnects, one for the memory and another for the ASP. These allowed us to split the interfaces of these two components across two different memories such that they each had access to both of the memories.

1. Configure *both* of the OpenBus Interconnect components **I_MCU** and **I_ASP** as it appears in the following dialog.

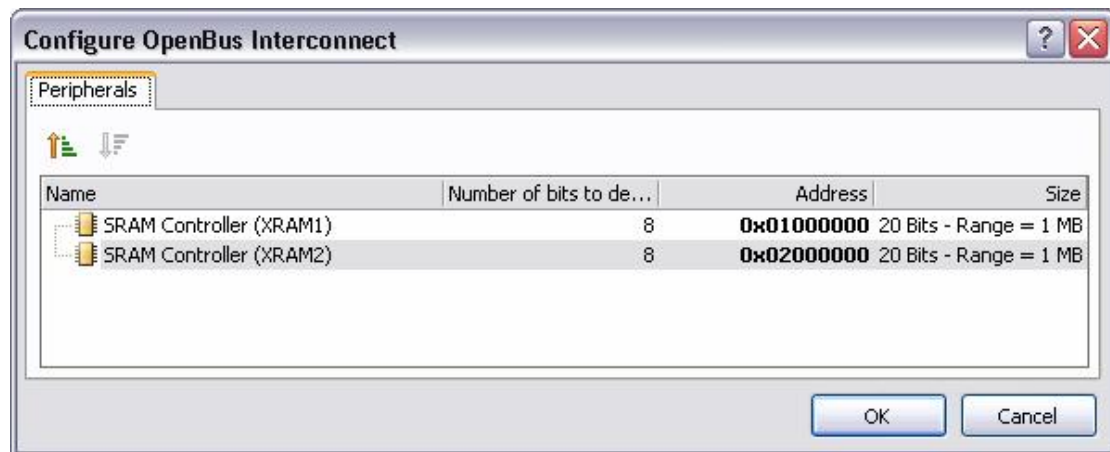


Figure 88 - Configuring the OpenBus Interconnect

2. Click OK to return to the OpenBus document.
3. Save your work.

13.16 Finalizing the OpenBus Portion of the Design

At this stage, we have nearly completed the OpenBus portion of our design and it should be obvious that by using OpenBus, the process of capturing your design becomes faster. Likewise it's important to recognize how much more accurate the connectivity can be when it's relegated to the software to sort out and not a human process.

As we complete the OpenBus portion of our design, one thing remains, and that is the configuration of the processor memory. If you recall from our schematic discussions, the processor memory was being auto-defined by a hardware.h file. We had selected this option rather than rely on the error-prone process of having to manually key in this data every time we made a change to our design.

Under OpenBus, this process is nearly identical and the next exercise will revisit this.

13.17 Exercise 41 – Configuring the Processor Memory

In this exercise we will configure the processor's peripheral and internal memories. We will also configure the processor to use the auto-generated hardware.h file, rather than try maintaining the memory configuration manually.

1. Right click the component **MCU** in the OpenBus document and select **Configure Processor Memory** to launch the **Configure Processor Memory** dialog.
2. Ensure that the **hardware.H (C Header File)** checkbox is checked and ensure that the memory configuration above appears as it does in Figure 89.

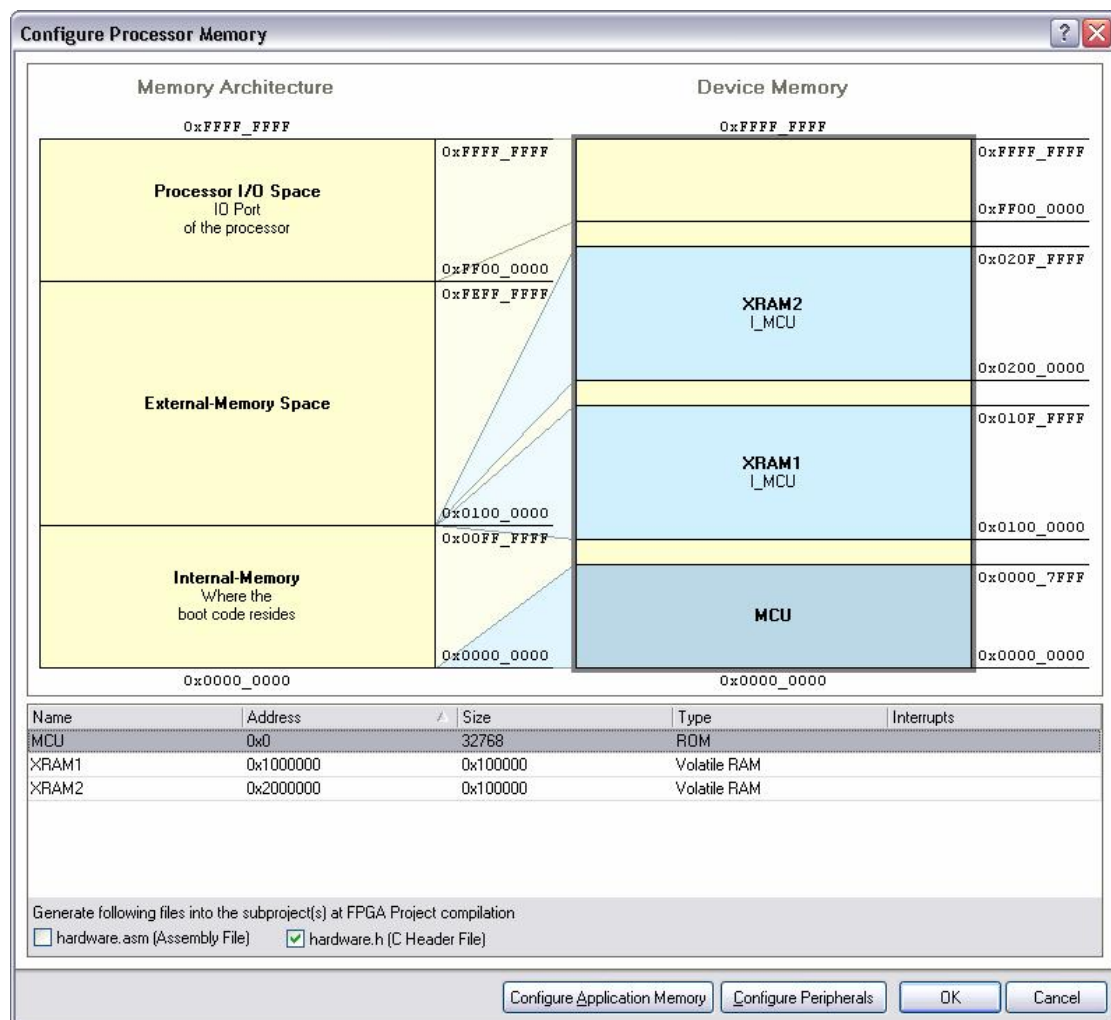


Figure 89. Configuring the processor memory

3. Select the **Configure Peripherals** button to launch the Configure Peripherals dialog.
4. Ensure that the **hardware.H (C Header File)** checkbox is checked and ensure that the memory configuration above appears as it does in Figure 90.

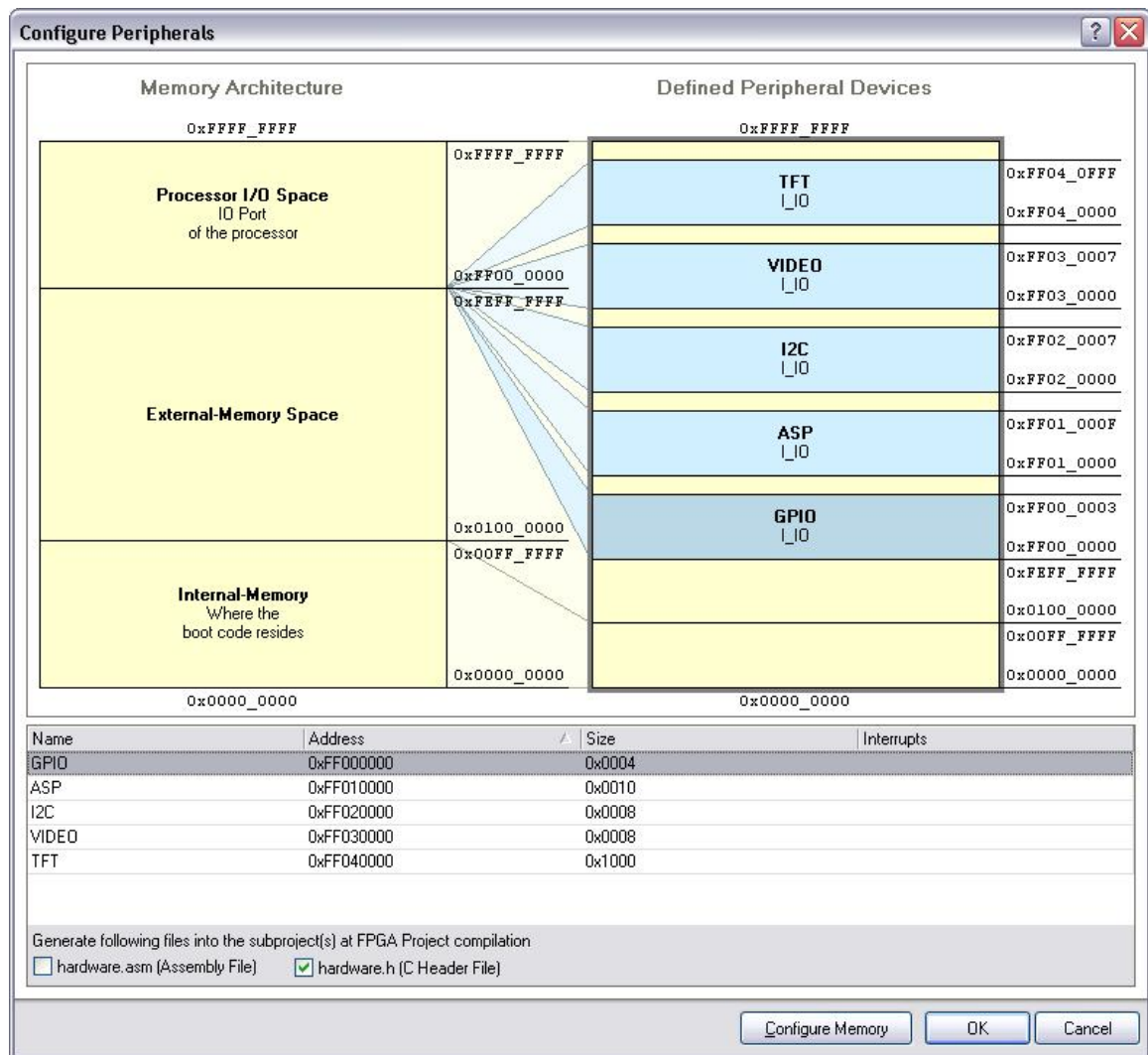


Figure 90. Configuring the processor peripheral memory

5. Click OK to return to the OpenBus document.
6. Save your work.

13.18 Exercise 42 - Creating a sheet symbol from an OpenBus Document

To link the OpenBus document to a parent schematic, you need to create a sheet symbol from the OpenBus document and place it on the parent schematic. In this exercise we will create sheet symbol from the open bus document.

1. Open Video_Capture_Schematic.SchDoc
2. Select **Design >> Create Sheet Symbol From Sheet or HDL**.
3. When the Choose Document to Place dialog box appears, select the Video_Capture_OpenBus.OpenBus document and click OK.
4. A large sheet symbol will be attached to the cursor. Position it at the center of the schematic page and click once to commit the placement.

The sheet entries on the newly placed sheet symbol have been loosely grouped with inputs on the left and outputs on the right. You must now go through a process of unraveling all of these sheet entries so that you can connect them to the port plugins on the NanoBoard more easily.

5. A sheet symbol with the various sheet entries properly organized is located in the **Snippets** panel as `OpenBus_Sheet_Symbol1`. If you intend to use this symbol be certain your filename matches the filename in the Sheet Symbol properties.
6. A snippet has also been created with the basic Port Plug In components, placed and ready to be wired into your sheet symbol. You can find this in the Snippets panel under listed as `Open_Bus_PortPlugIns`.

13.19 Exercise 43 – Linking an Embedded Project to an OpenBus Design

The Process of Linking an OpenBus FPGA Design to an Embedded Project is identical to the process with which we linked these two projects we'd created the design using a schematic-only flow.

1. From the Projects panel, switch to the Structure Editor mode.
2. Locate the embedded project `EmbeddedProject.PrjEmb` in the Valid Sub Projects and Configurations section at the bottom of the dialog.
3. Drag and drop the embedded project over MCU (TSK3000A) in the top section of the panel. This will link the two projects.

Note: Embedded projects can be linked to more than one FPGA project. This makes the process of porting code from one design to another quite easy.

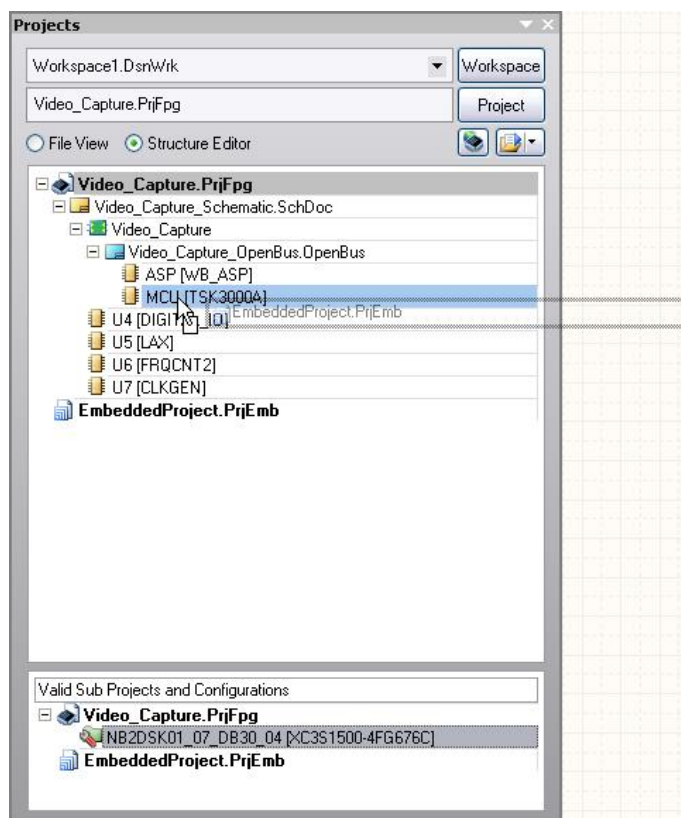


Figure 91 – Linking the two projects

4. Return to the File View mode in the Projects panel.
5. Save your work.

13.20 Exercise 44 – Processing an OpenBus Design

The Process of Constraining and Building an OpenBus FPGA Design is identical to the process we'd undergone when we'd built our design using a schematic-only flow. In this exercise we will revisit the process of constraining and building our design.

1. First we need to configure our design. With the FPGA and Embedded Projects linked, switch to the Devices View under **View»Devices Views** or by hitting the  button in the toolbar at the top of the screen.
2. Right Click the image of the Desktop NanoBoard at the top of the window in the **Devices View** and select **Configure FPGA Project»Video_Capture.PrjFpg**

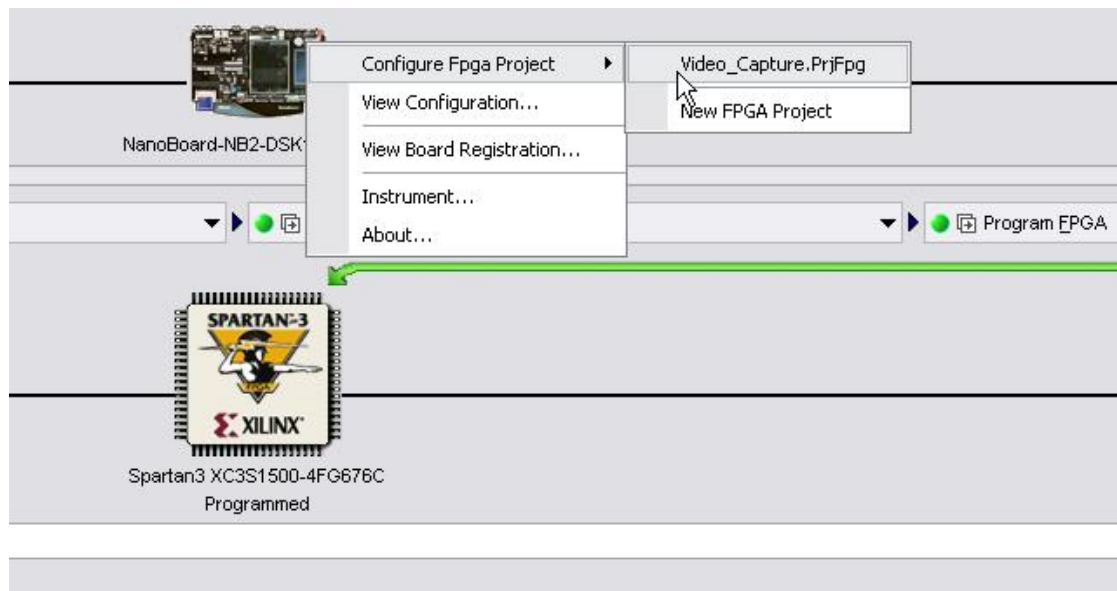


Figure 92 – Auto configure the OpenBus design

This will automatically configure the FPGA Project to include the constraint files required to target the hardware on the Desktop NanoBoard and will launch the **Configuration Manager For Video_Capture.PrjFpg** dialog.

3. Within the Configuration Manager, click the Add button to add a constraint file to the existing configuration.
4. Locate the MyConstraint.constraint file and add this to the current configuration (this file had been used by our previous schematic-only implementation of this project).
5. Locate MyConstraints.Constraint in the **Constraint Files** columns and check the box in the **Configurations** column to add it to the existing configuration.
6. Click OK to close this dialog and the Hard JTAG Chain should appear now in the main window. Likewise, the Soft, Nexus JTAG Chain should display the Host Processor MCU and our Virtual Instruments.
7. Save your work.
8. Locate the **Compile, Synthesize, Build, Program FPGA** buttons running left to right just below the Desktop NanoBoard icon. Click once on the words **Program FPGA** to begin the build process.

14 Review